F3

Fakulta elektrotechnická
Katedra mikroelektroniky

Diplomová práce

Návrh a implementace výukového mikrokontroléru založeného na zásobníkovém procesoru

Viktor Bohuněk

leden 2023

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Bohuněk** Jméno: **Viktor** Osobní číslo: **474229**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra mikroelektroniky**

Studijní program: **Elektronika a komunikace**

Specializace: **Elektronika**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Návrh a implementace výukového mikrokontroléru založeného na zásobníkovém procesoru

Název diplomové práce anglicky:

Design and Implementation of Microcontroller with Stack Processor for Education

Pokyny pro vypracování:

Cíl práce je návrh a implementace mikrokontroléru se zásobníkovou architekturou. Při vypracování se řiďte následujícími pokyny:
1) Proveďte rešerši literatury a seznamte se s existujícími implementacemi zásobníkových procesorů.
2) Na základu rešerše navrhněte vlastní mikrokontrolér, který bude implementován na vhodném FPGA kitu.
3) K mikrokontroléru připravte potřebné programové vybavení tak, aby jej bylo možné používat ve výuce.
4) Zhodnoťte dosažené výsledky.

Seznam doporučené literatury:

[1] Stack computers: The new wave. Pittsburgh, USA: Ellis Horwood, 1989. ISBN 9780745804187.

Jméno a pracoviště vedoucí(ho) diplomové práce:

doc. Ing. Stanislav Vítek, Ph.D. katedra radioelektroniky FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **09.09.2022** Termín odevzdání diplomové práce: _____

Platnost zadání diplomové práce: **19.02.2024**

doc. Ing. Stanislav Vítek, Ph.D. podpis vedoucí(ho) práce	prof. Ing. Pavel Hazdra, CSc. podpis vedoucí(ho) ústavu/katedry	prof. Mgr. Petr Páta, Ph.D. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

. _____ Datum převzetí zadání

_____ Podpis studenta

Prohlášení autora

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č.121/2000 Sb., autorského zákona, ve znění pozdějších předpisů, zejména skutečnost, že České vysoké učení technické v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 citovaného zákona.

V Praze dne _____ podpis _____

Viktor Bohuněk

Anotace

Diplomová práce se zabývá návrhem tzv. *soft-core* mikrokontroléru postaveného na zásobníkovém procesoru. V teoretické části jsou tyto procesory představeny a definovány a jsou rozebrány vybrané existující projekty zásobníkových procesorů.

Praktická část práce se věnuje popisu navrženého mikrokontroléru a programového vybavení, které k němu náleží. Výsledný mikrokontrolér *EduStackMachine* kromě základních periferií obsahuje také hardwarovou ladicí jednotku schopnou komunikovat přes *JTAG* s programem *OpenOCD*. Pro pohodlnější vývoj programů je k dispozici jednoduchý jazyk symbolických instrukcí včetně překladače. V rámci práce byla také navržena jednoduchá deska plošných spojů, která přidává použitému kitu další periferie, a především přidává důležité komunikační sběrnice. V samotném závěru praktické části jsou zhodnoceny a porovnány dosažené výsledky s existujícími řešeními. V závěru jsou pak představeny další cesty možného budoucího vývoje.

I přes některé nedostatky byl cíl práce splněn a mikrokontrolér je ve stavu, kdy je možné jej používat ve výuce pro demonstraci fungování zásobníkových procesorů.

Klíčová slova

zásobníkový procesor, mikrokontrolér, *soft-core*, assembler, *FPGA*, *DE10Lite*

Annotation

The diploma thesis deals with the concept of a soft-core micro-controller with a stack processor core. In the theoretical part, these processors are introduced and defined, also, a selection of some already–existing stack–processor projects are presented.

The practical part of the thesis is concerned with the description of the implemented micro-controller as well as the programming equipment which belongs to it. The final micro-controller *EduStackMachine*, aside of basic peripherals, contains hardware–debugging unit which is able to communicate through *JTAG* with *OpenOCD* debugger. For more comfortable development of the programs, there is a simple assembly language, including its corresponding assembler, available. A simple printed circuits board was designed, which adds a few more external peripherals to *FPGA* board and, moreover, it further expands communication interfaces. At the very end of the practical part of the thesis the results were evaluated and compared with pre-existing solutions. In the conclusion of the thesis, some possible ways of further research are proposed.

Despite some deficiencies, the aim of the thesis was met and the micro-controller is in state, where it could be used during lessons and/or for demonstrating principles of stack processor.

Keywords

stack processor, micro-controller, *soft-core*, assembler, *FPGA*, *DE10Lite*

Poděkování

Děkuji panu doc. ing. Stanislavu Vítkovi, Ph.D. za vedení mé diplomové práce. Mé poděkování dále patří rodině a přátelům, kteří mě po celou dobu podporovali.

Obsah

Seznam zkratek a symbolů

Zkratka	Celý název
ALU	Arithmetic Logic Unit
ASIC	Application Specific Integrated Circuit
CMOS	Complementary Metal–Oxide–Semiconductor
CPLD	Complex Programmable Logic Devices
DWM	Domain Wall Memory
ESM	EduStackMachine
Ethernet	Souhrn technologií pro počítačové sítě standardizované v IEEE 802.3
FPGA	Field Programmable Gate Array
GCC	GNU Compiler Collection
GPIO	General-purpose input/output
I^2C	Inter-Integrated Circuit
IEEE	Institute of Electrical and Electronics Engineers
IoT	Internet of Things
ISA	Instruction Set Architecture
JTAG	Rozhraní definované standardem IEEE 1149.1, případně IEEE 1149.7
LED	Light Emitting Diode
MOSFET	Metal Oxide Semiconductor Field Effect Transistor
OpenOCD	Open On-Chip Debugger
RPN	Reverse Polish Notation
TCP/IP	Sada protokolů používaná v síti internet
VHDL	Very High Speed Integrated Circuit Hardware Description Language
LAB	Logic Array Block
SRAM	Static Random Access Memory
KiB	Jednotka označující 1024 bytů, na rozdíl od kB označující 1000 bytů
UART	Universal asynchronous receiver-transmitter
MIPS	Million instructions per second
R	Přístup pouze pro čtení
R/W	Přístup pro čtení a zápis
W	Přístup pouze pro zápis, při čtení vrací 0
WARL	Je možné zapsat libovolnou hodnotu, ale čtena bude pouze platná hodnota
SEASM	StackEduAssembler language
JSON	JavaScript Object Notation

Seznam obrázků

Seznam tabulek

Seznam zdrojových kódů

Kapitola 1

Úvod

První stroj klasifikovatelný jako analogový počítač sestrojil kolem roku 1820 anglický inženýr Ch. Babagge. Jeho mechanický počítač *Difference engine* počítal hodnoty pro námořní navigační tabulky. *Difference engine* a jeho následovníci ale neměli měnitelný program, protože ten byl dán jejich konstrukcí. Pro každý druh problému tedy bylo nutné navrhnout nový počítač.

Po druhé světové válce byl dokončen počítač *ENIAC*[1]. Jednalo o první elektronický počítač, který bylo, opět zcela poprvé, možné naprogramovat. Přesto se však programování *ENIACu* příliš nelišilo od jeho mechanických předchůdců, jelikož program byl definován propojkami a přepínači, které bylo nutné ručně nastavit tak, jak daný program vyžadoval. Zlom nastal až s počítačem *Manchester baby*, který měl program uložený v paměti. Od této chvíle započal dodnes trvající závod o sestavení co nejvýkonnějšího, zároveň ale co nejmenšího a nejméně energeticky náročného počítače.

V 60. letech byla zahájena komerční produkce tranzistorů a také vznikly první prototypy integrovaných obvodů založených právě na tranzistorech. V 70. letech pak na trh vstupuje technologie *CMOS*[2] založená na *MOSFETech*[3]. A již na konci tohoto desetiletí bylo možné touto technologií vyrobit první jednočipový procesor *Intel 4004*. Od té doby je vývoj digitálních obvodů úzce spojen právě s technologií *CMOS*.

V oněch 70. letech se zároveň objevily hlasy, které navrhovaly radikální změnu v konstrukci procesorů. Jejich záměrem bylo nahradit registry zásobníkem. Od tohoto kroku si slibovaly vznik jednodušších, menších, ale stále výkonných procesorů vhodných především do prostředí, které vyžaduje spolehlivost a rychlé zpracování dat. První, kdo tyto hlasy vyslyšel, byla společnost *Novix*, která na trh uvedla 16bitový procesor *NC4000*, později přeznačený na *NC4016*. Tento procesor na hardwarové úrovni podporoval jazyk *Forth*[4], který u něj nahradil assembler[5]. Postupně bylo na trh uvedeno ještě několik dalších zásobníkových procesorů. Vzhledem k jejich odlišné architektuře ale bylo nutné nákladně a složitě upravovat existující kompilátory a další nástroje. Tyto přidané náklady společně se stále levnější výrobou integrovaných čipů nakonec zapříčinily konec těchto procesorů a koncem 90. let se již žádné nevyráběly, což trvá dodnes.

Ovšem s příchodem *IoT*[6] znovu nabírá na síle poptávka po malých, úsporných, ale zároveň výkonných procesorech. *IoT* zařízení jsou obvykle konstruována tak, aby je bylo možné nainstalovat a bez další údržby provozovat minimálně několik let. Aby toho bylo možné docílit, jsou dnes běžně napájena chemickým článkem s dlouhou životností nebo nově také pomocí energie okolí (tzv. *energy harvesting*). Oba způsoby si žádají maximální omezení příkonu procesoru v každé fázi – během chodu programu, přechodu do hibernace/spánku a při probouzení z hibernace/spánku. Zásobníkové procesory postavené na nových typech nevolatilních pamětí (např.

[1] Electronic Numerical Integrator And Computer
[2] Complementary Metal–Oxide–Semiconductor
[3] Metal Oxide Semiconductor Field Effect Transistor
[4] Interpretovaný programovací jazyk, který využívá pro ukládání operandů zásobník
[5] Jazyk symbolických adres
[6] Internet of Things

DWM[7]) mohou být cestou, jak všechny tyto podmínky snadno a efektivně splnit. [18] [11]

V teoretické části budou stanoveny požadavky na mikrokontrolér vhodný pro výuku principů této neobvyklé architektury. Budou také představeny projekty jiných *soft-core* zásobníkových procesorů. V praktické části bude popsán mikrokontrolér vlastního návrhu založeného na zásobníkovém procesoru – *EduStackMachine*. Kromě hardwarové implementace je součástí práce jazyk symbolických instrukcí *SEASM* a jeho překladač. V závěru práce pak budou vyhodnoceny dosažené výsledky a dána doporučení pro další rozvoj.

[7] Domain Wall Memory

Kapitola 2

Teoretická část

V rámci kapitoly budou definovány základní pojmy z oblasti zásobníkových procesorů, představen seznam požadavků na výsledný mikrokontrolér a představeny a zhodnoceny již existující projekty postavené na zásobníkových procesorech.

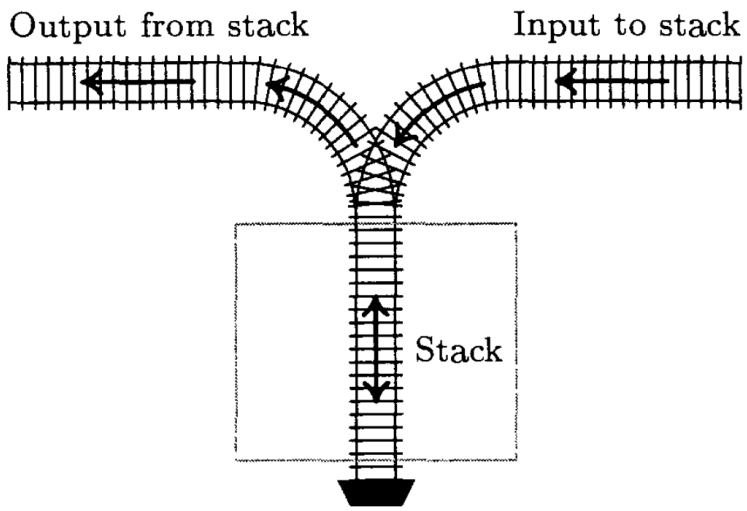
2.1 Definice zásobníku

„Zásobník lze definovat jako speciální případ tzv. lineárního seznamu, jenž je sekvencí n prvků kdy $n \geq 0$ $X[1], X[2], \ldots, X[n]$, jejichž jedinou strukturální vlastností je relativní pozice mezi ostatními prvky v seznamu, jako by byly všechny seřazeny přímo za sebou. Dále platí, že pokud $n > 0$, pak $X[1]$ je prvním prvkem lineárního seznamu a $X[n]$ je posledním. Dále pokud $n > 0$, tak platí že pro libovolné číslo k z intervalu $1 < k < n$ je pro prvek $X[k]$ předcházejícím prvkem $X[k-1]$ a následujícím prvkem $X[k+1]$.

Nad lineárním seznamem je možné definovat mnoho operací, pro zásobník jsou podstatné tyto:

- čtení (případně úprava) hodnoty k-tého prvku lineárního seznamu,
- vložení nového prvku před libovolný k-tý prvek lineárního seznamu,
- smazání libovolného k-tého prvku lineárního seznamu,
- zjištění počtu prvků uložených v lineárním seznamu.

Zásobník je takovým lineárním seznamem, u kterého všechny operace čtení, úprav, mazání a vkládání prvků probíhají pouze na jeho konci. Speciálním případem zásobníku je pak takový, u kterého je povoleno čtení libovolného k-tého elementu.“ (Převzato z [10], přeložil V. Bohuněk)



Obrázek 2.1: Ilustrace fungování zásobníku jako železnice; převzato z [12].

2.2 Definice zásobníkového procesoru

Zásobník patří k běžné výbavě prakticky každé moderní procesorové architektury. Aby bylo možné jednoznačně definovat zásobníkové procesory, vytvořil P. Koopman, americký odborník na vestavné systémy, v *Stack computers: The new wave* [10] taxonomii procesorů založenou na těchto parametrech:

- počet v hardware implementovaných zásobníků (jeden = **S**ingle, více = **M**ultiple)
- velikost těchto zásobníků (malé = **S**mall, velké = **L**arge)
- počet specifikovaných operandů instrukcí (0, 1, 2)

Všechny variace poskytují celkem 12 kategorií, do kterých lze zařadit prakticky každou existující architekturu. Kategorie jsou pojmenovány jednoduše počátečními písmeny anglických slov a číslicí odpovídající počtu operandů.

Kategorie ML0 (**vícero velkých** v hardware implementovaných zásobníků s instrukcemi s **0** operandy) označuje právě zásobníkové procesory. Tato kombinace vlastností se v praxi projevuje tak, že procesor pro operace, které vyžadují operandy (matematické operace, manipulace s pamětí), je bere ze zásobníku a není tak nutné tyto operandy specifikovat. Pro zápis takových operací se používá *RPN*[1], rozdíl mezi touto notací a běžně používanou, tzv. infixovou notací, je zachycen v rovnicích (1) a (2). Ty zároveň ilustrují největší výhodu *RPN*, a to, že je jednoznačná i bez použití závorek, protože operace se vždy aplikuje na dva předchozí operandy, které jsou nahrazeny výsledkem, který může okamžitě vstoupit jako operand do další operace.

$$(3 - 4) \times 5 = -5 \tag{1}$$

$$3 \quad 4 - 5 \times = -5 \tag{2}$$

2.3 Seznam požadavků na mikrokontrolér

Aby bylo možné posoudit vhodnost již existujících procesorů, je nutné sestavit seznam základních požadavků.

[1] Reverse Polish Notation, česky Postfixová notace

Jádrem mikrokontroléru musí být zásobníkový procesor, jak je definován výše (2.2).

Vzhledem k tomu, že mikrokontrolér bude sloužit jako demonstrační pomůcka a také jako platforma pro řešení laboratorních cvičení a domácích úloh, bude potřeba mít dobrý přístup k internímu stavu procesoru. Mikrokontrolér tedy musí být vybaven ladicí jednotkou s takovým rozhraním a programovým vybavením, aby bylo možné ladit na něm běžící programy s pomocí běžného počítače.

Vzhledem k určení mikrokontroléru jakožto vzdělávací pomůcce bude vhodné upřednostnit méně efektivní, zato snadno pochopitelný design nad efektivním, ale komplikovaným. Především je nutné vyvarovat se *pipeliningu*, tedy techniky souběžného rozpracování více instrukcí najednou. Zároveň bude vhodné omezit instrukční sadu na minimum nutných, jednoduchých, instrukcí, za cenu nižší efektivity výsledných programů. Vhodné také bude aby, instrukční sada poskytovala prostor pro budoucí rozvoj.

Aby bylo možné výsledný design považovat za mikrokontrolér, je nutné aby byly k dispozici alespoň tyto periferie:

- vstupně/výstupní piny
- *UART*[2], který bude zajišťovat komunikaci uživatelských programů s počítačem.

Pro smysluplné využití mikrokontroléru bude potřeba, aby bylo možné měnit jeho program po naprogramování jeho designu do *FPGA*. Opět je důležité, aby toto bylo možné provést z běžného počítače.

Pro procesor musí také existovat *toolchain*[3], který kromě výše uvedených funkcí zajistí také překlad z jazyka symbolických adres na strojový kód.

Poslední, však neméně důležitou součástí,K je dokumentace všech částí, která umožní pochopit jak mikrokontrolér funguje, jak pro něj psát programy a jak nastavit a používat jeho *toolchain*.

2.4 Rozbor vybraných existujících zásobníkových procesorů

Cílem této sekce je představit již existující zásobníkové *soft-core* procesory a vyhodnotit zda jsou vhodné pro použití v rámci této práce.

Vzhledem k tomu, že se zásobníkové procesory už přes 30 let komerčně nevyrábí, se komunita kolem této neobvyklé architektury uchýlila k používání stále levnějších programovatelných logických obvodů, především *FPGA*[4] a simulátorů. Nezřídka za novým zásobníkovým procesorem stojí firmy, které těží z malé velikosti a vysokého výkonu těchto procesorů. Vícero nových designů je také spjato s programovacím jazykem *Forth*, který opět nahrazuje *assembler*.

2.4.1 *ZPU*

V roce 2008 uvolnila společnost Zylin jako otevřený software pod licencí *BSD* svůj 32bitový zásobníkový procesor *ZPU*. Referenční implementace je napsána v jazyce *VHDL*[5]. *ZPU* je navržen s ohledem na minimální využití prostředků *FPGA*. Pro *ZPU* existuje *fork*[6] *GCC*[7], takže je možné snadno a rychle pro něj vyvíjet software. V rámci *FPGA* je možné *ZPU* použít k ladění dalších obvodů nebo pro implementaci logiky, kterou nemá smysl implementovat přímo

[2] Universal asynchronous receiver-transmitter

[3] Soubor programů, které umožňují vývoj programů pro konkrétní platformu

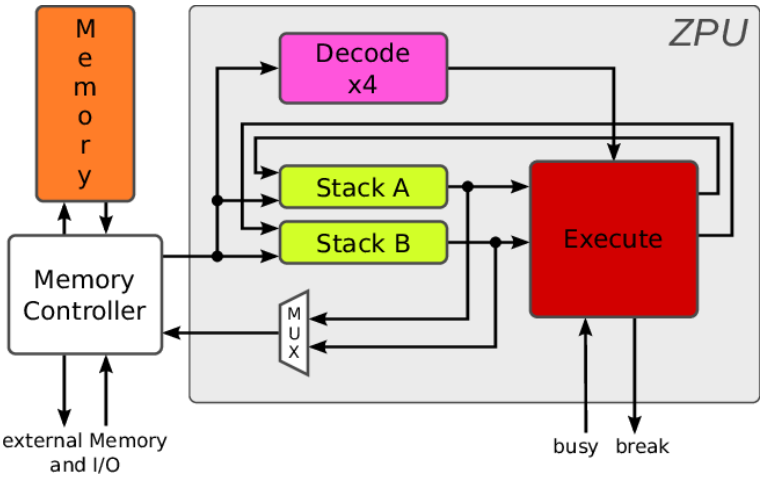
[4] Field Programmable Gate Array

[5] Very High Speed Integrated Circuit (VHSIC) Hardware Description Language

[6] Označení pro alternativní větev vývoje projektu, nezávislou na vývoji původního projektu; v oblasti Open-Source se jedná o častý jev

[7] GNU Compiler Collection

v hardware. Na obr. 2.2 je zachycen celý procesor *ZPU*, který se skládá celkem ze tří jednotek (zásobníky A a B se počítají jako jedna jednotka). [6] [17]



Obrázek 2.2: Architektura *ZPU* na úrovni bloků; převzato z [17]

K dispozici jsou dvě referenční implementace – *small* a *medium*. Varianta *small* používá pouze 16bitovou datovou sběrnici (ostatní jsou stále 32bitové), a také implementuje pouze nutnou část instrukční sady přímo v hardware. Zbytek je emulován pomocí ostatních instrukcí. Větší a výkonnější varianta *medium* používá všechny sběrnice 32bitové a většina instrukcí je implementována v hardware. Nejlepší konfigurovatelnost poskytuje implementace *Zealot*, která skrze parametrizaci designu dovoluje ovlivnit, které instrukce budou emulovány a které implementovány v hardware. Pro ZPU také existuje podpora operačních systémů *eCos*[8], *FreeRTOS* a *mikroCLinux*.

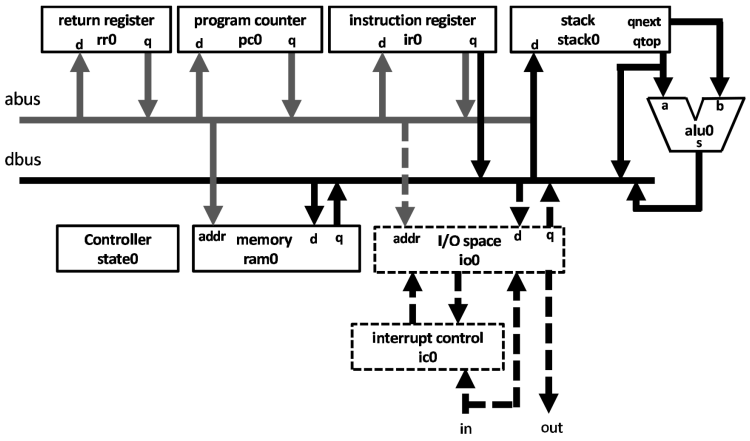
Použití *ZPU* jako základu mikrokontroléru není vhodné. Instrukční sada je ovlivněna požadavky kompilátoru *GCC*, a i když je možné ji kompletně implementovat v hardware, není to obvyklá cesta. Podpora pro periferie je mizivá, a kromě toho není konzistentní napříč implementacemi. Výrazným nedostatkem je také kvalita dokumentace, bez níž je obtížné pro studenta snadno pochopit, jak procesor funguje. Navíc, kvůli optimalizaci velikosti nemá procesor plnohodnotnou ladicí jednotku – je pouze schopen sestavit jednoduchý výpis adres ze zásobníku návratových adres při pádu programu. Pro praktické využití při výuce ještě chybí možnost nahrávat program do již naprogramovaného *FPGA*.

2.4.2 *TinyCSE*

TinyCPU je 16bitový procesor vyvinutý specificky pro účely výuky architektury počítačů na univerzitě v Hirošimě. *TinyCPU* není z pohledu dříve uvedené taxonomie zásobníkovým procesorem, protože používá pouze jeden v hardware implementovaný stack (viz obr. 2.3). Zjednodušení vychází z toho, že *TinyCPU* je během výuky postupně implementováno samostatně každým studentem a až v druhé polovině semestru jsou pro něj psány programy. Referenční implementace je napsána ve *Verilogu*. Celkem je k dispozici 28 instrukcí rozdělených na 9 řídicích a 19 výpočetních. Programy je možné psát v *assembleru* (*TINYASM*) a pseudo-C (*TINYC*), které se skládá z výběru klíčových slov jazyka *C*. [14], [13]

TinyCSE je rozšířením *TinyCPU* o řadič přerušení a jednotku mapující externí periferie do odděleného paměťového prostoru. Součástí *TinyCSE* jsou také nové periferie – časovač, kontrolér VGA, sériová linka a PS/2 rozhraní pro myš a klávesnici.

[8] Embedded Configurable Operating System



Obrázek 2.3: Architektura *TinyCSE* (přidané komponenty vyvedeny čárkovaně); převzato z [13]

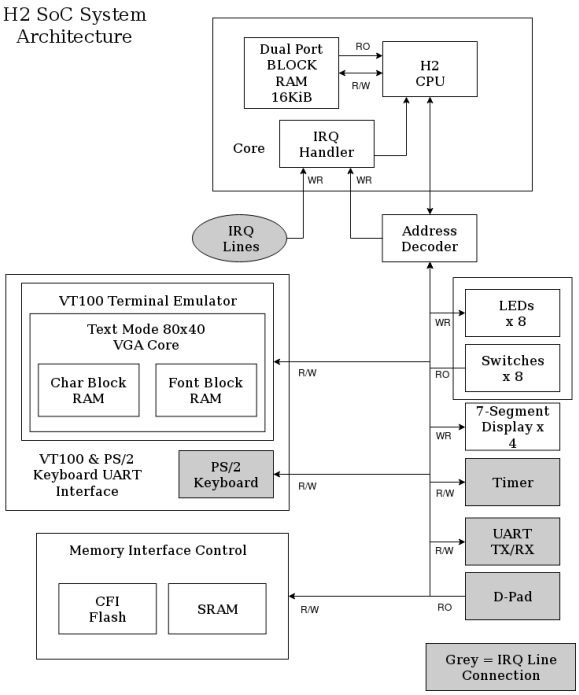
Hlavní překážkou pro použití *TinyCSE* je odlišná architektura, která využívá jen jeden zásobník jak pro operandy, tak návratové adresy. Dalšími jsou pak absence ladicí jednotky a možnosti nahrát program do již naprogramovaného *FPGA*. Dokumentace je velmi strohá, protože většinu práce na konkrétní implementaci odvede přímo student sám. Vzhledem k jeho původnímu určení a z něj vyplývajících omezení není *TinyCPU* vhodným základem pro budoucí mikrokontrolér.

2.4.3 *H2*

Jednodeskový 16bitový *Forth* počítač *H2* vyvinul během šesti let R. Howe jako rozšíření 16bitového *Forth* procesoru *J1* od J. Bowmana z roku 2010. Procesor J1 byl designován v jazyku *Verilog* jako řešení problému *streamovaní*[9] nekomprimovaného videa pomocí *TCP/IP*[10] přes *Ethernet*[11]. Pozoruhodné je, že *J1* dosahuje přibližně 100 *ANS Forth*[12] MIPS při taktovací frekvenci 80 MHz. [7], [2]

H2 je implementován v jazyce *VHDL* a je kompatibilní s *Forthem* pro *J1*. Oproti *J1*, který byl určen pouze jako pomocný procesor, je *H2* plnohodnotným jednodeskovým počítačem včetně hardwarového emulátoru terminálu *VT100* s výstupem na *VGA* a vstupem z *PS/2* klávesnice. Kromě toho přibyly ještě další standardní periferie – časovač, *UART* a řadič přerušení (viz obr. 2.4). Projekt také zahrnuje simulátor a *disassembler*[13] napsaný v jazyku C. *H2* není vybaven hardwarovou ladicí jednotkou, pro ladění programů lze použít simulátor.

[9] Kontinuální přenos audiovizuálního obsahu pomocí počítačové sítě
[10] Souhrnný název pro sadu protokolů používaných v síti internet
[11] Souhrn technologií pro počítačové sítě standardizované v *IEEE 802.3*
[12] Standardizovaná verze jazyka *Forth* z roku 1994
[13] Program, který z binárního spustitelného souboru zrekonstruuje kód v jazyce symbolických adres



Obrázek 2.4: Architektura *H2*; převzato z [7]

Instrukční sada *J1* se skládá z pouze 15 instrukcí a sada *H2* přidává dalších 6. Programy napsané ve *Forthu* pro *J1* se kompilují přímo na strojové instrukce, kdežto u *H2* je v procesoru zabudovaný interpret[14] *Forthu*.

H2, potažmo *J1*, jsou dotažené projekty s dobrou dokumentací a komunitou, která tuto architekturu stále zkoumá a dále rozvíjí (např. *Exploring the J1 Instruction Set and Architecture using SIMPL* [5]). Pro využití v rámci této práce ale *H2* i *J1* postrádá ladicí jednotku. Navíc, program *J1* nelze měnit po naprogramování *FPGA*. *H2* sice toto omezení nemá, ale tuto funkci zajišťuje již zmíněný interpret jazyku *Forth*, čímž v jistém ohledu porušuje požadavek na jednoduchou instrukční sadu.

[14] Počítačový program, který přímo vykonává zápis jiného programu bez nutnosti tento zápis nejprve přeložit do strojového kódu

Kapitola 3

Praktická část

V této sekci budou popsány jednotlivé implementované celky v rámci projektu *EduStackMachine*, zkráceně *ESM*. Cílem tohoto projektu je naplnit výše stanovené požadavky (2.3). Jádrem celého projektu je zásobníkový procesor vlastního návrhu doplněný o základní periferie a ladicí jednotku komunikující s *OpenOCD*[1] přes *JTAG*. Součástí projektu je také překladač pro *assembler* se základní podporou pro makra a návěští, aby se mohl programátor soustředit především na práci se zásobníkem.

3.1 Použité technologie

Při zpracovávání praktické části byla použita celá řada technologií, protože účelem této práce je popsat autorův postup a přínos, jsou použité technologie shrnuté v následujícím výčtu:

Intel Quartus Prime Integrované vývojové prostředí od společnosti Intel pro návrh obvodů pro programovatelná logická zařízení (*FPGA*, *CPLD*[2])

Sigasi Studio Integrované vývojové prostředí od společnosti Sigasi založené na OpenSource projektu Eclipse, jeho hlavní předností je základní kontrola návrhu v reálném čase a možnost integrace s Intel Quartus Prime, kdy Sigasi Studio slouží především jako mnohem lépe vybavený editor návrhu

Questa*-Intel Simulátor *HDL*[3] postavený na produktu ModelSim od společnosti Siemens, který má přidanou podporu pro programovatelné logické obvody od Intelu

OpenOCD OpenSource nástroj pro ladění programů na různých hardwarových platformách

Tcl Jednoduchý skriptovací jazyk přítomný v ostatních použitých nástrojích

Node.js OpenSource prostředí, které umožňuje spouštět skripty pro skriptovací jazyk *Javascript* mimo prohlížeč, slouží k vytváření serverových aplikací nebo okrajově pro vývoj nástrojů spouštěných z příkazového řádku

JTAG Rozhraní definované standardem *IEEE 1149.1*, případně *IEEE 1149.7*, je používané pro testování desek plošných spojů pomocí techniky tzv. *boundary scan*, v případě procesorů se pak využívá možnost nad *JTAG* definovat vlastní protokol pro komunikaci s interní ladicí jednotkou

FT2232HL Integrovaný obvod od společnosti FTDI, který je vybaven rozhraním *USB* 2.0 a dvěma jednotkami *MPSSE*[4], které mohou fungovat jako *transceivery* pro různé sběrnice

[1] Open On-Chip Debugger
[2] Complex Programmable Logic Devices
[3] Hardware Description Language
[4] Multi-Protocol Synchronous Serial Engine

Jest Testovací *framework*[5] pro *Javascript*

VHDL Jazyk pro návrh digitálních obvodů, určených pro programovatelná pole (*FPGA*, *CPLD*) nebo pro vytváření návrhu pro *ASIC*[6], definován ve standardu *IEEE 1076*

3.2 Cílová platforma

Po konzultaci s vedoucím práce byla jako cílová platforma zvolena deska *DE10-Lite* od společnosti terasIC (viz obr. 3.1). *FPGA* osazené na *DE10-Lite* je z rodiny Intel MAX10, konkrétně 10M50DAF484C7G.

Deska poskytuje vše potřebné pro provoz *FPGA*, tedy programátor s rozhraním *JTAG* přes *USB*[7], externí konfigurační paměť a generátor taktovacího signálu o frekvenci 50 MHz. Vzhledem k tomu, že se jedná o výukový kit, je k dispozici celkem 10 přepínačů, 2 tlačítkové spínače, 10 *LED*[8], 6 znaků sedmisegmentového displeje, akcelerometr s rozhraním I^2C[9], *VGA*[10] výstup s rezistorovou sítí pro generování signálů a napěťové sledovače pro *ADC*[11] zabudované v *FPGA*.

Desku je možné napájet přes *USB* konektor programátoru nebo přes konektor externího napájení. Pro připojení dalších periferií je k dispozici 40pinový konektor s roztečí pinů 2,54 mm a také konektory kompatibilní s Arduino UNO. Oba rozšiřující konektory poskytují napájení pro připojené periferie a to jak 3,3 V tak i přímo 5 V.

Přestože oba konektory poskytují 5 V, tak nejsou na desce osazeny žádné obvody, které by chránily piny *FPGA*, které mají maximální povolené stejnosměrné napětí 4,12 V a jsou kompatibilní s maximálně 3,3V logikou [9]. Naopak tlačítka a přepínače jsou k *FPGA* připojeny přes Schmittovy klopné obvody, aby se snížila šance na přechod interní logiky do metastabilního stavu.

Deska ovšem neposkytuje žádné komunikační rozhraní s počítačem připojeným přes *USB* konektor programátoru a *FPGA*. Tento problém řeší v rámci práce navržená deska využívající 40pinový rozšiřující konektor, popsaná v části 3.8.

[5] Předpřipravená sada nástrojů pro programovací jazyk, která zjednodušuje implementaci konkrétní funkcionality

[6] Application Specific Integrated Circuit

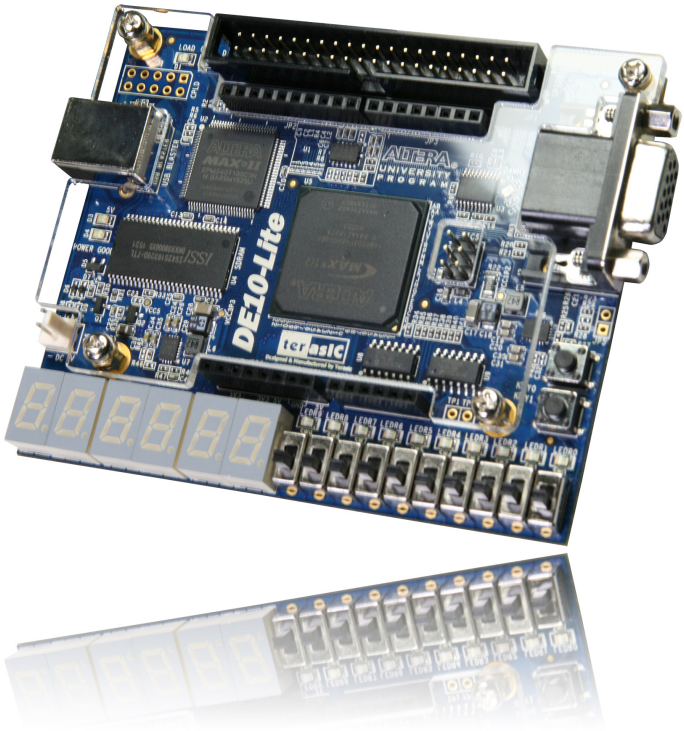
[7] Universal Serial Bus

[8] Light Emitting Diode

[9] Inter-Integrated Circuit

[10] Video Graphics Array

[11] Analog to Digital Converter



Obrázek 3.1: Fotografie *DE10-Lite*; převzato z [4].

3.3 Architektura a instrukční sada

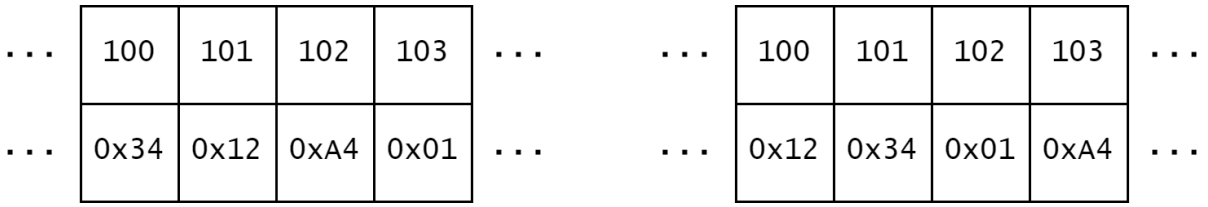
ESM je 16bitový zásobníkový mikrokontrolér *harvardské architektury*, má tedy fyzicky oddělenou programovou a datovou paměť. Šířka slova 16 bitů byla zvolena jako vhodný kompromis mezi praktickou uplatnitelností a složitostí architektury. *harvardská architektura* byla zvolena především protože značně ulehčuje návrh a zvyšuje jeho přehlednost. Tento přístup také značně snižuje nároky na kvalitu správy paměti a dovoluje programátorovi věnovat se více práci se zásobníkem.

Pořadí bytů v 16bitovém slovu je v rámci celého procesoru *little-endian*, tedy na vyšší adrese je uložen významnější byte, viz obr. 3.2. *ALU* zpracovává celá 16bitová čísla vyjádřená v přímé binární podobě z intervalu $\langle 0, 65535 \rangle$ a čísla vyjádřená v dvojkovém doplňku z intervalu $\langle -32768, 32767 \rangle$. Šířka adresové sběrnice je 13 bitů, tedy je možné adresovat $8192 = 8\text{k}$ pozic v paměti.

Instrukce mají proměnlivou délku od 1 po 3 byty, kdy první byte je vždy *OpCode*[12]. Vzhledem k povaze zásobníkových procesorů většina instrukcí nepotřebuje přímo specifikovat žádné operandy, a je tedy uložena pouze v jednom bytu. Šířka programové paměti byla zvolena jeden byte, vzhledem k proměnlivé délce to sice znamená, že se musí někdy číst více bytů pro vykonání jedné instrukce, ale na druhou stranu odpadají problémy se zarovnáním paměti při lichém počtu bytů, navíc je tak ušetřeno 8 signálů v rámci *FPGA* a snižuje se riziko, že dojdou dostupná propojení.

Instrukční sadu tvoří celkem 23 instrukcí, viz tab. 3.1. Podrobný popis instrukcí a jejich chování je v příloze této práce viz Dokumentace *EduStackMachine*. Vzhledem k tomu, že pro *OpCode* je vyhrazen celý byte, je aktuálně využito méně než 10 % maximálního počtu instrukcí. Mezery v použitých kódech jsou dány vývojem, kdy ze začátku byly operace *ALU* dány bity

[12] Číslo kódující konkrétní instrukci



(a) *Little-endian* (b) *Big-endian*

Obrázek 3.2: Příklad uložení čísel `0x1234` a `0x01A4` do paměti

daného *OpCode*. Tento přístup se ale ukázal jako nepřehledný a nepraktický. Aktuálně se pro nastavení operací *ALU* používá signál výčtového typu `alu_op_type`.

3.4 Implementace mikrokontroléru

Mikrokontrolér *EduStackMachine* je implementován v jazyce *VHDL-1993*. Všechny signály používají pozitivní logiku, tedy jako logická 1 se bere stav signálu 1 ve *VHDL*. Propojení mezi entitami jsou provedena v top entitě `edu_stack_machine`. Pro sdílené konstanty, funkce a definice komponent je pak v rámci pracovní knihovny `work` vytvořen balíček `edu_stack_machine_common`.

V celém návrhu je použitý jen jeden hodinový signál o frekvenci 1 MHz (dále jen `CLK`) odvozený od vstupního hodinového signálu 50 MHz pomocí fázového závěsu v entitě `system_pll`. Fázový závěs je implementován pomocí *megafunkce*[13] `altpll`, která zajistí správné použití prostředků *FPGA*.

V rámci mikrokontroléru existují 2 resetovací domény – `ndmreset`, která zahrnuje všechny komponenty mimo ladicí subsystém a `dmreset`, která je vyhrazená pro ladicí subsystém. Po inicializaci *FPGA* se zajistí reset všech komponent pomocí signálu `system_pll_locked`, který signalizuje, že už je k dispozici výstup fázového závěsu. Tento zdroj resetuje obě domény. Dalšími vstupy jsou pak tlačítka 1 a 2 na komunikační desce (viz obr. 3.6), kdy tlačítko 1 ovládá pouze doménu `ndmreset` a tlačítko 2 opět obě dvě. Aby bylo zajištěno, že reset bude proveden správně, je každý zdroj zvlášť synchronizován s pomocí entity `sync_bits_altera` z knihovny *PoC - Pile of Cores* [8] tak, aby přechod z 0 na 1 proběhl okamžitě, ale kritický přechod z 1 na 0, kdy dojde k uvolnění resetu, proběhl pouze na náběžnou hranu `CLK`. Toto opatření zabraňuje tomu, že by se některé entity aktivovaly ještě v době, kdy jiné jsou ještě stále v resetu.

Popis návrhu ostatních částí mikrokontroléru je rozdělen do celkem tří celků, které sdružují spolu související entity – procesor, subsystém pro ladění a periferie (3.4.1, 3.4.2, 3.4.3).

3.4.1 Procesor

Zásobníkový procesor, který je základem mikrokontroléru, je řízen řadičem `core_fsm`. Ten přijímá jednotlivé instrukce z programové paměti a zajišťuje jejich vykonání viz Zpracování instrukcí. Pro přehlednost je pro dekódované instrukce použit výčtový typ `instruction_type`. Konverze binárních kódů instrukcí na hodnoty `instruction_type` je řešena v entitě `instruction_decoder`, která je instantiována[14] přímo v `core_fsm`.

Další důležitou částí je *ALU*[15], která je implementována v entitě `alu`, která usměrňuje tok dat skrz procesor a provádí jednoduché matematické a logické operace s jedním nebo dvěma operandy. Výpočet hodnoty a příznaků probíhá v kombinatorickém procesu a výsledek je tak k dispozici hned po změně některého z operandů. Zdrojové signály operandů je možné měnit

[13] Označení systému na vytváření entit pro IP jádra v Intel Quartus Prime

[14] Instantiation = krok, který *VHDL* entitu nebo komponentu zakomponuje do nadřazeného celku, zároveň jsou definovány hodnoty parametrů a připojeny signály podřazené entity/komponenty

[15] Arithmetic logic unit

Tabulka 3.1: Přehled instrukční sady *ESM*

OpCode	Název	Popis
0x00	No operation	Žádná operace, „čekání“
0x01	Break point	Pokud je aktivní ladicí režim, zastaví vykonávání programu, jinak se chová jako NOP
0x02	16bit integer literal	Vloží 16bitové číslo do TOPD0
0x03	8bit integer literal	Vloží 8bitové číslo do TOPD0
0x04	Zero branch	Podmíněný skok na zadanou adresu, provede se pokud TOPD0 == 0
0x05	Non-zero branch	Podmíněný skok na zadanou adresu, provede se pokud TOPD0 != 0
0x06	Unconditional branch	Nepodmíněný skok na zadanou adresu
0x07	Call	Skok na zadanou adresu s uložením aktuální adresy do TOPR
0x08	Return	Skok na adresu v TOPR
0x09	Store	Zápis dat z TOPD0 na adresu z TOPD1 v datové paměti
0x0A	Load	Čtení dat z datové paměti na adrese z TOPD0 do TOPD0
0x0B	Peripherals write	Zápis dat z TOPD0 do registru periferie na adrese v TOPD1
0x0C	Peripherals read	Čtení dat z registru periferie na adrese v TOPD0 do TOPD0
0x0D	Discard	Zahodí slovo v TOPD0
0x0E	Duplicate	Duplikuje TOPD0
…		
0x20	Addition	TOPD1 = TOPD1 + TOPD0; v TOPD0 budou uloženy příznaky
0x21	Subtraction	TOPD1 = TOPD1 – TOPD0; v TOPD0 budou uloženy příznaky
0x22	Left-shift	TOPD0 = TOPD0 <<1; v TOPD0 budou uloženy příznaky
0x23	Right-shift	TOPD0 = TOPD0 >>1; v TOPD0 budou uloženy příznaky
…		
0x28	Bitwise NOT	TOPD0 = NOT(TOPD0); v TOPD0 budou uloženy příznaky
0x29	Bitwise AND	TOPD0 = TOPD0 AND TOPD1; v TOPD0 budou uloženy příznaky
0x2A	Bitwise OR	TOPD0 = TOPD0 OR TOPD1; v TOPD0 budou uloženy příznaky
0x2B	Bitwise XOR	TOPD0 = TOPD0 XOR TOPD1; v TOPD0 budou uloženy příznaky

pomocí dvou zabudovaných multiplexerů. K dispozici jsou v zásadě všechny datové signály přítomné v procesoru, výběr zdroje provádí řadič pro každý operand zvlášť. *ALU* je vybavena dvěma interními registry, jeden je určen pro `operand_b` a druhý pro uložení příznaků (`flags`). Registr pro příznaky je potřeba, protože výstup z *ALU* je sdílen mezi výsledkem a právě příznaky.

Příznaky jsou uloženy po každé matematické nebo logické operaci do *TOPD0*, vytlačí tedy výsledek operace co *TOPD1*. Celkem je k dispozici 5 různých příznaků, které umožňují programátorovi nahlížet na čísla, se kterými pracuje, jako na neznaménková nebo znaménková ve dvojkovém doplňku. Uspořádání příznaků v rámci slova je znázorněno na obr. 3.3 a je převzato z *ATmega328PB* [1]. Význam jednotlivých bitů:

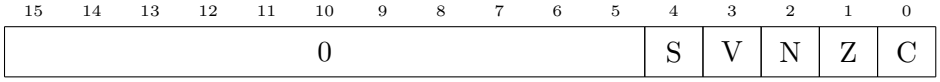
C – Carry; přetečení, příznak je nastaven na 1, pokud výsledek matematické operace překročil rozsah 16 bitů; při operaci bitového posunu je v něm uložen bit, který je vysunut mimo rozsah

Z – Zero; poslední výsledek libovolné operace byl 0

N – Negative; poslední výsledek matematické operace měl bit č. 15 nastaven na 1, tedy se ve dvojkovém doplňku jedná o záporné číslo

V – Two's complement overflow; při poslední matematické operaci došlo k přetečení dvojkového doplňku

S – Sign; určen jako exkluzivní logický součet (XOR) příznaku `N` a `V`, vyjadřuje jaké znaménko by měl mít výsledek poslední matematické operace



Obrázek 3.3: Příznaky poskytované *ALU*

Zásobníky jsou implementovány vlastní logikou a prošly několika fázemi vývoje, které měnily způsob ukládání dat a měnily i chování rozhraní s ohledem na potřeby procesoru. První verze byla implementována s pomocí *BRAM*[16], ale záhy se ukázalo, že zpoždění 2 hodinových cyklů, které *BRAM* v *FPGA* z rodiny MAX10 má, je příliš veliké a zdržuje vykonávání instrukcí. Tento problém by zřejmě šlo vyřešit pokročilejší mezipamětí, ale vzhledem k velikosti obou zásobníků se zdálo výhodnější přejít na použití *Distributed RAM*[17], která může fungovat plně asynchronně, a je tak možné čtení a zápisy na různé adresy provést bez přidaného zpoždění. Obecně je *Distributed RAM* nevhodná jen pro velké paměti, protože je velmi náročná na prostředky *FPGA*, a to jak samotné *LAB*[18], tak především propojení mezi nimi, což může v krajním případě vést k selhání kompilace celého návrhu.

Datový zásobník pojme 64 slov po 16 bitech a poskytuje kromě vrchního prvku (`TOPD0`) také ten po něm následující (`TOPD1`). Tyto prvky jsou kromě hlavní paměti uloženy ve zvláštních registrech uvnitř entity `data_stack`. Tyto registry jsou aktualizovány při každém zápisu nebo mazání. Oba prvky jsou trvale dostupné ke čtení. Všechny operace, které zásobník podporuje, jsou prováděny na náběžnou hranu `CLK`. Jaká operace bude provedena, určují stavy signálů `en` a `wr` viz tab. 3.2.

[16] Block RAM - bloky *SRAM* integrované v *FPGA*, určené pro ukládání většího množství dat
[17] Paměťové buňky jsou implementovány pomocí logických bloků *FGPA*
[18] Logic Array Block – nejmenší blok v rámci *FPGA* řady MAX10

Tabulka 3.2: Operace podporované datovým zásobníkem

wr	rd	Operace
0	0	Prvky jsou dostupné pro čtení
0	1	Vyjmutí vrchního prvku
1	0	Vložení nového prvku na vrchol
1	1	Přepsání vrchního prvku novými daty

Zásobník návratových adres je kratší a jednodušší variantou datového zásobníku. K dispozici je jen 16 slov po 16 bitech. Číst je možné vrchní prvek (`TOPR`) a chybí režim přepsání vrchního prvku viz tab. 3.3.

Tabulka 3.3: Operace podporované zásobníkem návratových adres

wr	rd	Operace
0	0	Prvky jsou dostupné pro čtení
0	1	Vyjmutí vrchního prvku
1	0	Vložení nového prvku na vrchol
1	1	Přednost dostane operace vyjmutí vrchního prvku

Programová paměť, která slouží jako zdroj instrukcí, je implementována pomocí *megafunkce* `altsyncram` v entitě `program_ram`. Použití *megafunkce* zajistí, že bude pro uložení dat použita *BRAM*, a navíc vystaví vhodné rozhraní pro komunikaci s pamětí. Paměť má z pohledu *ESM* (port A dvouportové *BRAM*) uspořádání 8×8k, tedy její kapacita je 8 KiB.

Adresu pro programovou paměť generuje entita `program_counter`. Adresa je uvnitř entity uložena v registru. Hodnota v registru je automaticky inkrementována na každou náběžnou hranu `CLK`, pokud je aktivní signál `en`. Naopak, pokud je aktivován signál `im`, dojde k přepsání hodnoty v registru hodnotou ze signálu `pc_im`.

Datová paměť v entitě `data_ram` je opět implementována s pomocí *megafunkce* `altsyncram`, data jsou tedy uložena v *BRAM*. Vzhledem k tomu, že data i adresa pro tuto paměť jsou zadávány pomocí výsledku z *ALU*, je její adresový registr vybaven funkcí podržení hodnoty. Paměť je pro *ESM* (port A dvouportové *BRAM*) strukturována jako $16 \times 8k$, což odpovídá kapacitě 16 KiB. Adresovat je možné pouze celá 16bitová slova.

Zpracování instrukcí

Zpracování instrukcí řídí stavový automat v entitě `core_fsm`, kompletní graf všech stavů a přechodů mezi nimi je součástí přílohy (obr. A.4). Automat po resetu procesoru vstoupí rovnou do stavu `FETCH`, který slouží jako zpoždění pro paměť, ale zároveň obsahuje i logiku rozhodování o přechodu do stavu `HALT`. Přechod se odehraje, pokud ladicí subsystém požádá o pozastavení chodu procesoru (`esm_core_halt_req`), nebo se provádí krokování programu (`esm_core_stepping`), nebo byla vykonána instrukce *Break point* (`break_inst`), anebo došlo při vykonávání jiné instrukce k výjimce (`esm_core_exception_internal`). Ze stavu `HALT` se může procesor vrátit do stavu `DECODE`, jen pokud nedošlo k výjimce a ladicí subsystém požádal o obnovení chodu programu (`esm_core_resume_req`). Ve stavu `DECODE` je už k dispozici aktuální *OpCode* z programové paměti a interní signál `instruction`, přivedený z entity `instruction_decoder`, již obsahuje aktuální dekódovanou hodnotu datového typu `instruction_type`. Podle signálu `instruction` se zvolí správná větev vykonávání instrukce a rovnou se provede první krok instrukce.

Ve stavech následujících po `DECODE` se provádí ostatní kroky v rámci dané instrukce a z posledního kroku každé instrukce se přejde do stavu `PRE_FETCH`, který slouží jako čekání na inkre-

mentaci PC a který zároveň nastaví všechny signály v rámci core_fsm do výchozího stavu. Tímto je ukončeno vykonávání aktuální instrukce a přechodem do stavu FETCH se zahájí vykonávání následující instrukce.

3.4.2 Subsystém pro ladění

Subsystém pro ladění se řídí otevřeným standardem *RISC-V External Debug Support* verze 0.13.2 [15] a je schopný komunikovat s *OpenOCD*. Implementace ve *VHDL* vychází ze dvou entit neorv32_debug_dtm a neorv32_debug_dm z projektu *NEORV32 RISC-V Processor* [16]. Tyto entity přesně kopírují architekturu ladicího systému pro *RISC-V*, který je velmi flexibilní, protože se musí přizpůsobit velmi různorodým implementacím a aplikacím procesoru *RISC-V*. V neorv32_debug_dtm je realizováno *DTM*[19], které přes externí rozhraní *JTAG* zprostředkovává přístup k interní sběrnici *DMI*[20], pro kterou je *DTM* řídicím členem, jež může být pouze jeden. Na sběrnici *DMI* jsou pak připojeny jednotlivé *DM*[21]. Ty jsou pak již připojeny k *harts*[22] podle potřeby daného systému.

Pro *ESM* bylo nutné změnit v neorv32_debug_dm implementovaný režim ladění. Původní implementace používala tzv. *Execution Based* režim, tedy ladicí software (*OpenOCD*) mohl spouštět na procesoru instrukce, a tak provádět operace s registry a pamětí a řídit tok programu. Protože *ESM*, samozřejmě, instrukce z *ISA RISC-V* nepodporuje, bylo potřeba přejít na druhý možný režim, tzv. *Abstract Command Based*. Z dostupných *Abstract Commands* je implementován jen ten pro přístup k registrům, přičemž jeho implementace maskuje neexistenci *RISC-V hart* tím, že ignoruje zápisy do neimplementovaných registrů, a naopak vrací hodnoty i pro registry, které v *ESM* neexistují, ale bez kterých by *OpenOCD* nedokázal fungovat (registr MISA[23]). Kromě toho je pro čtení i zápis přes vyhrazené rozhraní v entitě program_counter zpřístupněn PC, což dovoluje *OpenOCD* před obnovením chodu programu změnit místo, od kterého se bude pokračovat. Specifickým případem je registr DCSR[24], který je implementován v rámci DM, namísto v procesoru. V tomto registru se nachází bity, které řídí chod core_fsm, tedy žádosti a potvrzení zastavení či obnovení chodu procesoru zmíněné v 3.4.1.

Další úpravou bylo přidání podpory pro přímý přístup do paměti pomocí *SBA*[25]. *SBA* počítá s jedním paměťovým prostorem adresovaným 32bitovou adresační sběrnicí. Protože pro paměti v *ESM* je potřeba maximálně 14bitová adresační sběrnice, bylo možné do tohoto prostoru umístit paměti tak, aby byly na rozumných adresách. Z dostupných 32 bitů je použito pouze 18. Bity 17–16 slouží k rozlišení paměti a ze zbylých 16 bitů se využije tolik, kolik je potřeba. Mapa paměťového prostoru je v tab. 3.4.

V kontextu *RISC-V* obchází *SBA* správu paměti jednotlivých *hartů* a přistupuje k paměti z globálního pohledu. Přístup do pamětí a zásobníků v *ESM* je ovšem řešen vyhrazeným rozhraním, které je nezávislé na zbytky *ESM*. Obě paměti mají k tomuto účelu aktivní druhý port B, na kterém poskytují šířku slova 16 bitů (u programové paměti se tedy liší od šířky slova, kterou má k dispozici *ESM*), a také je k dispozici signál byteena, který dovoluje operovat s jednotlivými byty ve slovu. V DM je pomocí byteena umožněno adresovat jednotlivé byty, a přistupovat tak k programové paměti stejně jako *ESM*. Režim přístupu po 16 bitech je potřebný pro *OpenOCD*, které jej využívá při dávkovém zápisu nebo při dávkovém čtení paměti. Zásobníky přístup přes SBA implementují vyhrazeným rozhraním, které má vlastní ukazatel do jejich vnitřní pamětí. Toto rozhraní ale nepodporuje adresování jednotlivých bytů. Díky oddělenému ukazateli je však možné číst nebo zapisovat do libovolného prvku v zásobníku. Zápis do paměti i zásobníků je

[19] Debug Transport Module
[20] Debug Module Interface
[21] Debug Module
[22] Hardware thread
[23] Obsahuje informace o implementovaných rozšířeních instrukční sady a základní parametry dané implementace *RISC-V hart*
[24] Debug control and status register
[25] System Bus Access

Tabulka 3.4: Mapa adresového prostoru z pohledu `DM`

Adresa počáteční	koncová	Paměť	Poznámka
`0x00000000`	`0x00001FFF`	Programová paměť	
`0x00002000`	`0x0000FFFF`	–	
`0x00010000`	`0x00013FFF`	Datová paměť	Z pohledu ladicího subsystému je uspořádána jako 8 × 16k, tedy má 14 adresových vodičů místo 13 z pohledu *ESM*
`0x00014000`	`0x0001FFFF`	–	
`0x00020000`	`0x00020080`	Datový zásobník	Adresován po bytech, stejně jako datová paměť
`0x00020081`	`0x0002FFFF`	–	
`0x00030000`	`0x00030020`	Zásobník návratových adres	Adresován po bytech, stejně jako datová paměť
`0x00030021`	`0xFFFFFFFF`	–	

ale možný jen v případě, že stavový automat v `core_fsm` je ve stavu `HALT`, předchází se tak nekonzistencím způsobeným zápisem na stejnou adresu v jeden okamžik.

Pro přístup k vybraným registrům a signálům *ESM* je využita možnost definovat v rámci adresového prostoru registrů *RISC-V* uživatelské registry, které jsou pak po nakonfigurování *OpenOCD* dostupné během ladění. Tímto způsobem jsou zpřístupněny pro čtení vrchní prvky zásobníků (`TOPD0`, `TOPD1`, `TOPR`), interní ukazatelé zásobníků (tedy stav jejich naplněnosti) a signál `exception` z `core_fsm`.

3.4.3 Periferie

Řadič periferií implementovaný v entitě `peripherals_ctrl` má několik vnitřních registrů, které odpovídají příslušným externím pinům a jsou k němu připojeny fronty *UART*[26]. Směrem ke zbytku procesoru je vystaveno rozhraní s adresovou sběrnicí o šířce 4 bitů, datové sběrnice a signály a signály `wren` a `rden`, které určují operaci. Adresy jednotlivých registrů a popis přístupu k nim je v tab. 3.5. Dostupné periferie a jejich přístupnost pro čtení a zápis pak odpovídají výbavě desky *DE10-Lite*, viz obr. 3.5. Fungování řadiče periferií z pohledu programátora je detailněji popsáno v příloze práce Dokumentace *EduStackMachine*.

Většina externích pinů má předem daný směr určený typem připojené periferie. Piny připojené k *LED* nad spínači a k segmentům displejů jsou vždy výstupy. Naopak, piny připojené ke spínačům a tlačítkům jsou vždy vstupy. Pravými *GPIO*[27] piny jsou tedy pouze ty, jež jsou vyvedené na rozhraní `Arduino 16`.

Vstupní piny jsou synchronizovány pomocí synchronizátoru v entitě `sync_bits_altera` z knihovny *PoC - Pile of Cores* [8]. Použití této entity by mělo zabránit propagaci případného metastabilního stavu.

GPIO piny jsou implementovány pomocí obousměrných třístavových budičů. Směr každého pinu je nastavitelný hodnotou v registru `arduino_en`, kdy 0 znamená, že pin je ve stavu vysoké impedance a je z něj čtena hodnota. I tyto piny jsou (v případě, že jsou nastaveny jako vstupní) synchronizovány stejně jako ostatní vstupní piny.

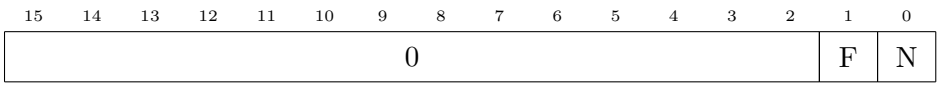
Implementace *UART* je provedena tak, aby bylo maximálně jednoduché tuto komunikaci používat. Konfigurace parametrů přenosu je pevná (9600 baud, 1 start-bit, 8 datových bitů a 1

[26] Universal asynchronous receiver-transmitter

[27] General-purpose input/output

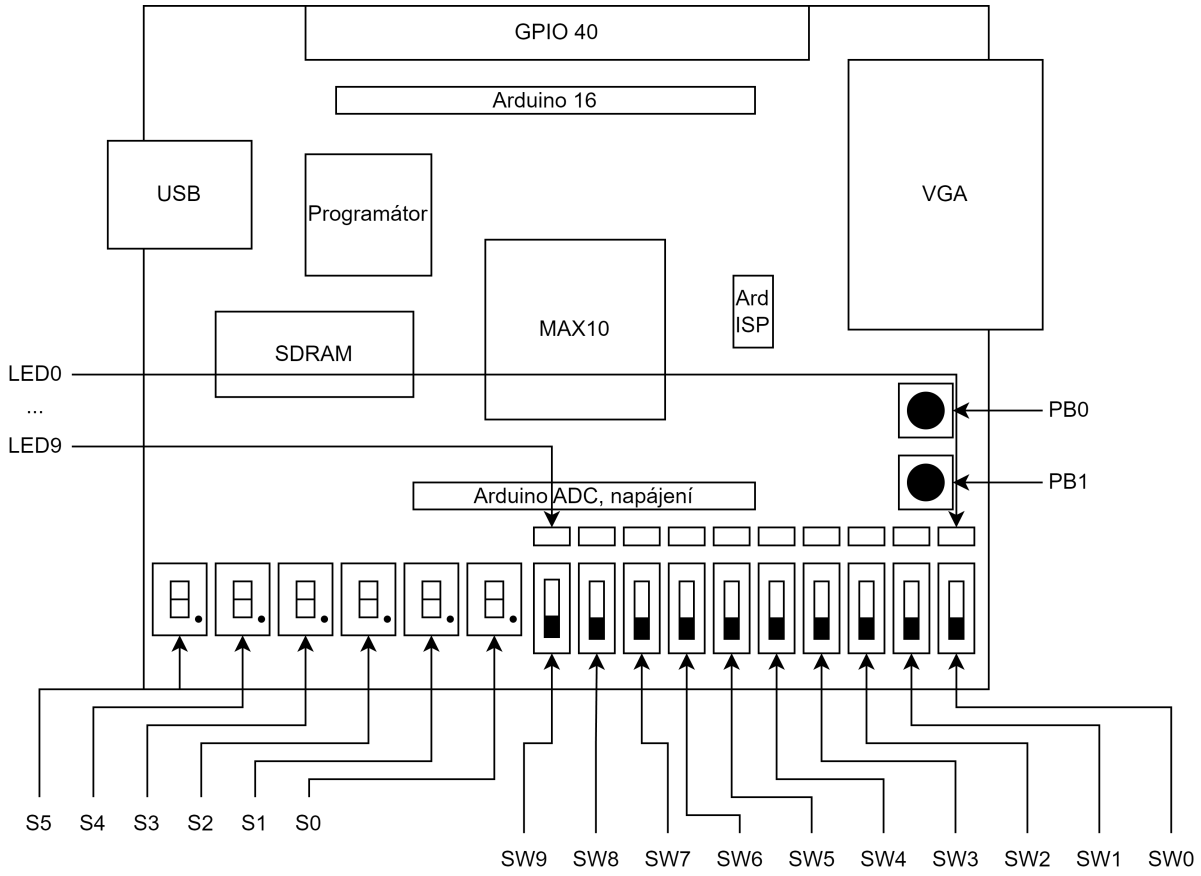
Tabulka 3.5: Seznam registrů řadiče periferií a jejich adres

Adresa	Název	Přístup
0x0	`arduino_in`	WARL
0x1	`arduino_out`	R/W
0x2	`arduino_en`	R/W
0x3	`led_out`	R/W
0x4	`segment_0_out`	R/W
0x5	`segment_1_out`	R/W
0x6	`segment_2_out`	R/W
0x7	`sw_pb_in`	WARL
0x8	`uart_in`	WARL
0x9	`uart_out`	W
0xA	`uart_status`	WARL



Obrázek 3.4: Registr `uart_status`

stop-bit), a aby nebylo nutné čekat na příchozí komunikaci, a blokovat tak vykonávání programu, má entita `uart` zabudované dvě fronty. Tyto hardwarové fronty jsou napojené na entitu `uart` z projektu *Simple UART for FPGA* [3]. Obě fronty jsou instantiovány z entity `fifo_128_byte`, která byla vygenerována pomocí *megafunkce*. Pro ukládání dat se používá *BRAM* a pro snažší realizaci registru pro přístup z *ESM* je fronta nakonfigurována tak, aby na jejím výstupu byl vždy dostupný prvek, a signál `rden` pak funguje jako potvrzení, že byl aktuální prvek přečten, a může tak být nahrazen dalším prvkem v pořadí. Tento režim významně zjednodušil rozhraní řadiče periferií. K této periferii ještě náleží registr `uart_status`, viz obr. 3.4. Bit `N` je 1, pokud v příchozí frontě čeká alespoň jeden prvek. Bit `F` je nastaven na 1, pokud je zaplněna odesílací fronta. Čtení z prázdné příchozí fronty nebo zápis do plné odchozí fronty nevytvoří výjimku, ale dojde k přečtení 0, respektive zapsaná data se ztratí.



Obrázek 3.5: Přehled periferií na desce *DE10-Lite*

3.5 Testování *VHDL*

V projektu jsou k dispozici 2 testy formou *VHDL testbenche*[28] – `alu_tb` a `core_fsm_tb`. Testy slouží pouze k základnímu ověření, že mikrokontrolér funguje korektně v *RTL*[29] simulaci. *Gate-level* simulace nebyla prováděna, protože bylo možné návrh rovnou testovat na *FPGA*. Výhodou tohoto přístupu je rychlost, protože *Gate-level* simulace je velmi časově náročný proces.

`alu_tb`

V tomto *testbenchi* se testují především příznaky, které *ALU* počítá pro operace sčítání a odečítání. Tyto příznaky jsou kritické pro korektní práci s čísly v rámci *ESM*. Tento *testbench* není složitý, je v něm instantiována pouze entita `alu`, a na její vstupy jsou pak přiváděny potřebné signály v procesu `stim_proc`, ve kterém jsou zároveň kontrolovány výstupy pomocí funkce `assert`.

`core_fsm_tb`

Testbench `core_fsm_tb` je postavený kolem entity `core_fsm` a pracuje na základě jednoduchého programu, který je nutné na začátku běhu simulace nahrát do programové paměti. K entitě `core_fsm` jsou pak připojeny další entity tak, aby vznikl prakticky celý procesor a bylo možné

[28] Označení pro speciální entitu, která obalí testovanou logiku a umožní ji nezávisle na zbytku návrhu testovat
[29] Register-transfer level

```
1  # Jednoduchy program na ukazku fungovani ESM
2  FUNCTIONS:        # Zde jsou vypsany vsechny funkce
3    function output # Funkce zapise 0x1234 na datovy zasobnik (DS)
4      IM16 0x1234   # Instrukce vlozi 16bitové cislo do DS
5      RET           # Instrukce pro navrat z rutiny
6    end function
7  PROGRAM:          # Blok instrukci, ktere tvori hlavni program
8    :main_loop      # Navesti pojmenovane "main_loop"
9    CALL output     # Skok do vyse definovane funkce "output"
10   DIS             # Instrukce zahozeni vrchniho prvku DS
11   UB main_loop    # Nepodmineny skok na navesti "main_loop"
```

Zdrojový kód 3.1: Ukázkový program v SEASM

na něm spustit testovací program. V rámci tohoto *testbenche* jsou ke `core_fsm` připojeny i jinak nevyužité signály `esm_core_state` a `esm_core_instruction`, které slouží pro řízení chodu procesu `stim_proc`.

Otisk paměti s programem je automaticky zkopírován při spuštění simulace z Intel Quartus Prime do složky, ze které je spouštěna simulace. Autor bohužel nedokázal přijít na to, jak zároveň automaticky spustit krátký *Tcl* skript, který zajistí nahrání dat do paměti. Skript je spolu se souborem otisku paměti uložen v projektu ve složce `core_fsm_tb_files` a je ho potřeba po spuštění Questa*-Intel ručně zkopírovat do příkazové řádky.

Pokud proběhne inicializace paměti v pořádku, nebude během vykonávání testu vypsána žádná chybová hláška. O jejich případné generování se opět stará proces `stim_proc`, který tentokrát pracuje v závěsu za stavovým automatem v `core_fsm`. Proces je řízen pomocí čekání na stav `DECODE` a `FETCH`. Ve stavu `DECODE` se pomocí funkce `assert` ověří, že byla dekódována očekávaná instrukce. Ve stavu `FETCH`, který znamená, že byla ukončena aktuální instrukce, se opět pomocí funkce `assert` testují vybrané signály, zda jejich hodnoty odpovídají očekávaným hodnotám po vykonání předchozí instrukce.

3.6 Jazyk symbolických instrukcí a překladač

Pro *ESM* byl navržen velmi jednoduchý jazyk symbolických instrukcí – *SEASM*, který dovoluje psát programy v přehlednější formě textových souborů. Jeho výstupem je binární soubor, který lze přímo nahrát pomocí *OpenOCD* a ladicího subsystému do programové paměti a spustit ho.

SEASM má velmi jednoduchou syntax, většina definovaných klíčových slov totiž odpovídá přímo konkrétním instrukcím, viz tab. 3.6. Pro ulehčení práce programátorovi jsou přidána návěští a funkce, na které se jde odkázat v příslušných instrukcích, a vyhnout se tak plánování toho, jak bude program uložený v programové paměti. Podporované je zadávání čísel v binární (`0b`), decimální (bez prefixu) a hexadecimální (`0x`) soustavě, ale pouze čísla decimální mohou mít znaménko mínus, které signalizuje, že má být hodnota uložena jako dvojkový doplněk.

Každý program se skládá pouze z jednoho souboru, který má pevnou strukturu. Povinně musí obsahovat obě dvě sekce – `FUNCTIONS` a `PROGRAM` – v tomto pořadí. Instrukce jsou psány do těchto sekcí (resp. do funkcí v bloku `FUNCTIONS`) každá na samostatný řádek s případným argumentem odděleným mezerou, viz zdrojový kód 3.1.

3.6.1 Překladač

Překladač *SEASM* je napsaný v jazyce *Javacript*, přesněji pro jeho verzi běžící v příkazové řádce – *Node.js*. Balíčky jsou spravovány systémem *yarn* a automatické testy jsou napsány ve *frameworku Jest*. Na přehlednost kódu dohlíží balíček *eslint*, s mírně upravenou verzí standardních

Tabulka 3.6: Přehled klíčových slov *SEASM*

Klíčové slovo	Popis
`:název_návěští`	Dvojtečka označuje vytvoření návěští, neprovádí žádnou operaci; nesmí existovat více ukazatelů na stejnou pozici
`ADD`	Instrukce – TOPD1 = TOPD1 + TOPD0, v TOPD0 budou uloženy příznaky
`AND`	Instrukce – TOPD0 = TOPD0 AND TOPD1, v TOPD0 budou uloženy příznaky
`BP`	Instrukce – Pokud je aktivní ladicí režim, zastaví vykonávání programu, jinak se chová jako NOP
`CALL název_funkce`	Instrukce – Skok na zadanou instrukci
`DIS`	Instrukce – Zahodí slovo v TOPD0
`DUP`	Instrukce – Duplikuje TOPD0
`end function`	Povinné ukončení každé funkce, neprovádí žádnou operaci
`function název_funkce`	Deklarace funkce, neprovádí žádnou operaci
`FUNCTIONS`	Blok funkcí, pouze jeden v celém souboru
`IM16 16_bitové_číslo`	Instrukce – Vloží 16bitové číslo do TOPD0
`IM8 8_bitové_číslo`	Instrukce – Vloží 8bitové číslo do TOPD0
`LOAD`	Instrukce – Čtení dat z datové paměti na adrese z TOPD0 do TOPD0
`LSHIFT`	Instrukce – TOPD0 = TOPD0 <<1, v TOPD0 budou uloženy příznaky
`NOP`	Instrukce – Žádná operace, „čekání“
`NOT`	Instrukce – TOPD0 = NOT(TOPD0), v TOPD0 budou uloženy příznaky
`NZB název_návěští`	Instrukce – Podmíněný skok na zadanou adresu, provede se pokud TOPD0 != 0
`OR`	Instrukce – TOPD0 = TOPD0 OR TOPD1, v TOPD0 budou uloženy příznaky
`PERIFRD`	Instrukce – Čtení dat z registru periferie na adrese v TOPD0 do TOPD0
`PERIFWR`	Instrukce – Zápis dat z TOPD0 do registru periferie na adrese v TOPD1
`PROGRAM`	Blok instrukcí hlavního programu, pouze jeden v celém souboru
`RET`	Instrukce – Vyskočení z rutiny
`RSHIFT`	Instrukce – TOPD0 = TOPD0 >>1, v TOPD0 budou uloženy příznaky
`STORE`	Instrukce – Zápis dat z TOPD0 na adresu z TOPD1 v datové paměti
`SUB`	Instrukce – TOPD1 = TOPD1 – TOPD0, v TOPD0 budou uloženy příznaky
`UB název_návěští`	Instrukce – Nepodmíněný skok na zadanou adresu
`XOR`	Instrukce – TOPD0 = TOPD0 XOR TOPD1, v TOPD0 budou uloženy příznaky
`ZB název_návěští`	Instrukce – Podmíněný skok na zadanou adresu, provede se pokud TOPD0 == 0

pravidel. Kód je rozdělen po logických celcích do tříd, které jsou uložené v souborech s názvem třídy. Tyto soubory mají příponu názvu `.mjs`, protože se jedná o moduly dle specifikace *ES6*. Pro logování do příkazové řádky se používá balíček *pino* a *pino-pretty*, který formátuje *JSON*[30] výstup z *pino* do lidsky čitelné podoby.

SEASM se spouští zadáním příkazu `yarn seasm název_soubor.asm`, výstupem je binární soubor s názvem vstupního souboru, ale s příponou názvu `.bin`. Překlad probíhá ve vícero krocích. V prvním kroku jsou všechny řádky ze vstupního souboru převedeny na objekty typu `AsmLine` a ty, které obsahují kód, jsou uloženy pro další zpracování. Během vytváření objektu jsou z každého řádku odstraněny nejprve komentáře, a potom netisknutelné znaky kolem kódu. V dalším kroku se hledají bloky `FUNCTIONS` a `PROGRAM`. Objekty *AsmLine* jsou roztříděny podle toho, do kterého bloku patří. Pak následuje zpracování bloku `FUNCTIONS`, kdy se projdou všechny objekty `AsmLine` z tohoto bloku a hledají se v něm funkce. Každá funkce musí začínat řádkem `function název_funkce` a končit `end function`. Objekty `AsmLine` mezi těmito řádky jsou předány jakožto parametr konstruktoru třídy `EsmProgram`, který z řádků vytvoří objekty třídy `EsmProgramToken`, které už představují konkrétní instrukce nebo návěští. Po zpracování funkcí je proveden stejný proces jako s vnitřkem funkce s celým blokem `PROGRAM`. V tento moment už je jasné, kolik bytů bude potřeba na vyjádření jednotlivých funkcí i hlavního programu a jsou k dispozici seznamy návěští, včetně jejich relativní pozice v rámci jednotlivých `EsmProgram` objektů a globální seznam funkcí. Posledním krokem je vygenerovat binární reprezentaci programu a doplnit chybějící adresy návěští a funkcí. Do binárního souboru je nejprve umístěn hlavní blok, jeho první instrukce je tedy na adrese 0. Za něj se pak postupně seřadí všechny funkce. Během procesu se určí, jaký offset od počátku má která funkce a je tak možné nahradit relativní adresy jejich návěští absolutními. Výsledkem je spustitelný binární soubor, který stačí nahrát do programové paměti.

Přiložené testy jsou rozděleny podle jednotlivých kroků zpracování programu. Testovací scénáře v `BlockParsing.test.mjs` mají ověřit, zda funguje správně hledání hlavních bloků `FUNCTIONS` a `PROGRAM`. V souboru `FunctionsParsing.test.mjs` jsou scénáře zaměřené na hledání funkcí a v `ProgramParsing.test.mjs` pak dochází k ověření správného zpracování klíčových slov a argumentů. Poslední scénář je uložen v `BinGen.test.mjs`, kde je rovněž uložen zdrojový kód 3.1 a jeho výsledná binární podoba tak, aby ji bylo možné porovnat s výstupem ze *SEASM*.

3.7 *Toolchain*

Toolchain pro *ESM* se skládá z již popsaného překladače jazyka symbolických instrukcí *SEASM* a OpenSource ladicího nástroje *OpenOCD*. Komunikace mezi tímto nástrojem a *ESM* probíhá přes rozhraní *JTAG*, které je dnes obvykle realizováno převodníkem na *USB*, a je tak možné jej používat s běžným počítačem po nainstalování vhodných ovladačů. Postup nastavení *OpenOCD* je popsán v příloze Návod na zprovoznění *OpenOCD*.

Konfigurační skript *OpenOCD* je uložen v `esm_openocd/esm_ftdi.cfg`. V první části je provedena konfigurace rozhraní, je zvolen konkrétní typ *JTAG* adaptéru a jsou nastaveny parametry přenosu. Následuje konfigurace parametrů *RISC-V hart*. Protože specifikace ladicího rozhraní není vždy v souladu s tím, jak *ESM* funguje, jsou v poslední části skriptu přidány funkce usnadňující práci s *ESM*. Nejprve jsou přidány uživatelské registry, aby s jejich hodnotou mohlo *OpenOCD* pracovat. Potom jsou přidány funkce zpracovávající oznámení o různých událostech v ladicím systému. Konkrétně jsou takto zpracovávány 3 různé události – obnovení vykonávání programu (*resumed*), krokování programu (*stepped*) a zastavení vykonávání programu (*halted*, *debug-halted*). U prvních dvou událostí je uživatel informován pouze výpisem do konzole, v případě zastavení programu je ještě přidán důvod, kterým může být buďto žádost o zastavení z *OpenOCD*, instrukce Break point, nebo hardwarová výjimka. V poslední části

[30] JavaScript Object Notation

skriptu jsou přidány funkce s jednotným prefixem `esm_`, které poskytují upravené rozhraní pro komunikaci s *ESM*. První sada těchto funkcí slouží k přirozenějšímu přístupu k uživatelským registrům, ve kterých jsou k dispozici vrchní prvky zásobníků a jejich ukazatelé. Druhá sada pak slouží k manipulaci s daty uvnitř zásobníků. Jak bylo popsáno v sekci 3.4.2 u hardwarové implementace ladicího subsystému, počítá se s tím, že všechny paměti jsou adresovatelné po jednotlivých bytech, ovšem implementace zásobníků tento režim nepodporuje. Přidané funkce tak překládají adresy z rozsahu $\langle 0, 63\rangle$ a $\langle 0, 15\rangle$ na odpovídající adresy dle tab. 3.4. Podrobný rozbor všech příkazů je v Dokumentace *EduStackMachine*.

OpenOCD umí pracovat s pamětmi také v dávkovém režimu. Je tak možné zkopírovat libovolný binární soubor do kterékoliv paměti přístupné přes ladicí subsystém a, samozřejmě, také uložit obsah paměti do binárního souboru. Příkaz dávkového zápisu se také využívá k nahrávání programu mikrokontroléru.

Díky správné implementaci hardwarové části specifikace je podporována i nepovinná funkce `reset halt`, která provede restart procesoru, a ještě před vykonáním první instrukce zastaví program. Toto je možné díky správně rozděleným resetovacím doménám (viz sekce 3.4) a vhodnému návrhu stavového automatu řadiče v `core_fsm`. Výsledkem je možnost ladit program od úplného počátku a ze známého stavu.

3.8 Komunikační deska

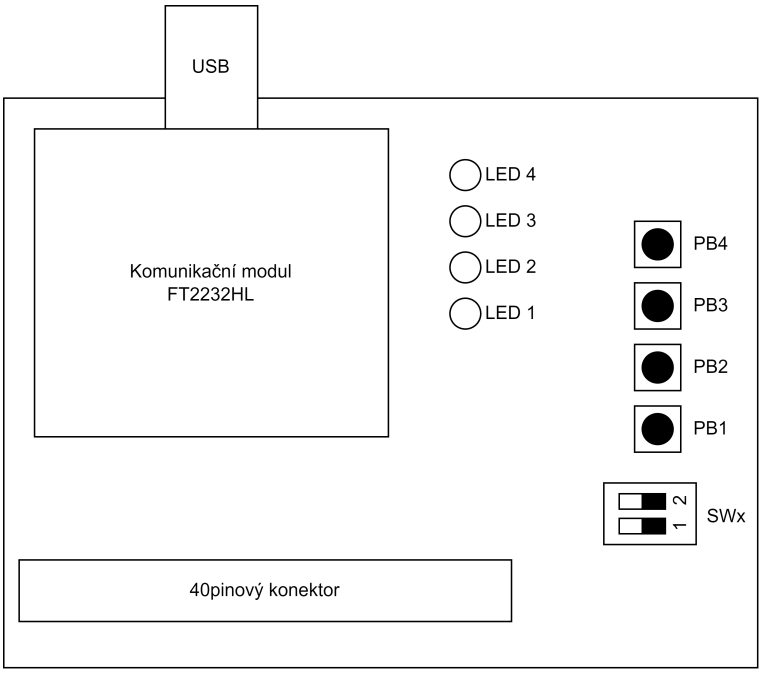
Úkolem této přídavné desky je připojit *FPGA* k převodníku na *JTAG* a sériovou linku; přidat několik dalších *LED*, přepínačů a spínačů; umožnit napájení desky *DE10-Lite* z *USB* konektoru převodníku. Vzhledem k tomu, že tato deska není těžištěm práce a v době návrhu desky nebylo možné čip FT2232HL od FTDI (nebo podobný) pořídit, je deska postavena kolem již hotového modulu převodníku (fialová deska na obr. A.3).

Pro připojení k *DE10-Lite* se používá 40pinový rozšiřující konektor a plochý kabel. Toto řešení by mělo omezit šanci, že se při manipulaci mechanicky poškodí konektor na *DE10-Lite*.

Aby bylo bezpečnější použít 5 V z modulu převodníku pro napájení *DE10-Lite*, jsou na desce použity stejné ochranné obvody jako na samotném *DE10-Lite* (viz obr. A.2). První ochranou jsou diody `D1`, `D2` a `D3`, které brání proudu téct do napájecího zdroje. Druhou ochranou je pak jednoduchý obvod, který funguje jako přepěťová ochrana, která při překročení napětí přibližně 5,7 V za diodami odpojí pomocí tranzistoru `Q1` napájecí zdroj. Odpojení závisí na prahovém napětí přechodu emitor-báze tranzistoru `T1` a zenerovu napětí `D4`, resp. je součtem těchto napětí. Pokud je tranzistor `T1` rozepnutý, udržuje tranzistor `Q1` v sepnutém stavu rezistor `R14`.

Přepínače a tlačítka jsou ke vstupům *FPGA* připojeny skrz Schmittovy invertující klopné obvody, aby se dosáhlo co nejstrmějšího přechodu při sepnutí/rozepnutí a minimalizovala se tak pravděpodobnost přechodu logiky v *FPGA* do metastabilního stavu.

Návrh prototypu počítal s tím, že deska bude vyrobena jako jednovrstvá, a tak jsou SMD komponenty osazeny na spodní straně, viz motiv plošného spoje v příloze A.1. Toto uspořádání není vhodné pro běžné použití ve výuce, navíc je deska podstatně větší než je potřeba. Další problém představuje konektor *USB* A, který používá komunikační modul. Mnohem vhodnějším konektorem je *USB* B, kterým je vybavena i deska *DE10-Lite*. I přes tyto nedostatky prototyp posloužil svému účelu – zajistil spolehlivé propojení *FPGA* a komunikačního modulu a přidal další vstupy a výstupy, viz obr. 3.6. Pro potřeby výuky bude vhodnější navrhnout zcela novou desku, která jmenované problémy odstraní.



Obrázek 3.6: Umístnění a číslování prvků na komunikační desce

Kapitola 4

Vyhodnocení

První krokem práce bylo provedení rešerše v oblasti zásobníkových procesorů. Výchozím bodem této rešerše pak byla kniha *Stack computers: The new wave* [10]. V knize jsou zásobníkové procesory představeny ze všech úhlů, ale tím nejpřínosnějším byl rozbor několika existujících architektur, ze kterého bylo možné čerpat inspiraci pro vlastní řešení. Problém představovalo hledání novějších projektů, které by ideálně byly také implementovány pomocí *HDL*, a byly tedy víc podobné výsledku této práce. Nejdůležitější je v tomto směru vývojová větev kolem procesoru *J1*, který vznikl relativně nedávno a vznikla kolem něj malá, ale vytrvalá komunita, která tento původně velmi jednoúčelový procesor dál rozvíjela. Zajímavou odbočkou pak byl *TinyCSE*, který se sice používá ve výuce, ale slouží k výuce návrhu hardwaru nikoli programování. *TinyCSE* je tak jednoduchý, že jej v rámci předmětu na Hirošimské univerzitě implementuje každý student samostatně. *ZPU* je pro změnu architektura, kterou po letech interního vývoje uvolnila komerční společnost Zylin pod OpenSource licencí, i kolem něj se vytvořila komunita, která stále pracuje na jeho vylepšeních. Rešerše vlastně prokázala, že zásobníkové procesory se sice už dekády nevyrábí, ale stále žijí svým životem a nachází uplatnění i v komerčních aplikacích.

Hlavní náplní práce byl návrh nového mikrokontroléru *ESM* a jeho *toolchainu*. Nejprve ale byla navržena instrukční sada, která pak byla implementována do procesoru, kolem kterého byl postaven jednoduchý *soft-core* mikrokontrolér. Přestože *ESM* nedosahuje žádných oslnivých parametrů (jednoduchým výpočtem vede na hodnotu 0,2 MIPS při taktovací frekvenci 1 MHz), tak to není nutné považovat za nikterak velký problém. V teoretické části popsané architektury používají pevnou délku instrukcí a spoustu optimalizačních triků, aby omezily úzká hrdla a dosáhly výkonů, kterými je tato nezvyklá architektura známá. Samozřejmě ale s optimalizací jde ruku v ruce složitější implementace a často také omezení, která blokují další vývoj, aniž by byla porušena zpětná kompatibilita. Instrukční sada *ESM* je se svým „plýtváním“ bity opakem přístupu ostatních architektur, ale pro další rozvoj tak zůstává ještě 233 volných *OpCodes*, a přitom nebude potřeba složitě přemýšlet jak změnit funkci, kterého bitu, aby bylo možné vmanipulovat do instrukční sady nové instrukce.

Návrh byl proveden v jazyce *VHDL-1993* v prostředí Intel Quartus Prime. Toto prostředí integruje všechny potřebné komponenty pro logickou syntézu, kompilaci *bitstreamu*[1] pro programování *FPGA* a simulaci s pomocí Questa*Intel. Autor jako jeho největší slabinu shledává kvalitu integrovaného editoru. V editoru totiž chybí mnoho funkcí, které ulehčují editaci kódu, především pak podpora pro více kurzorů. Dalším problémem je absence i jen základní kontroly napsaného kódu. Odhalení i těch nejtriviálnějších chyb tak proběhne až po několika desítkách sekund běhu kompilace, která se tím přeruší, takže ostatní chyby nejsou ani ohlášeny a odhalí je až další spuštění kompilace. Za vůbec nejhorší ale autor považuje funkci označení všech výskytů právě vybraného slova, která při kopírování vloží do schránky všechny označené výskyty. Tato pomocná funkce se sice dá vypnout v nastavení, ale občas se samovolně aktivovala i poté, a to

[1] Binární konfigurační soubor pro *FPGA*, jsou v něm zakódovány instrukce pro nastavení *LAB* a jejich propojení

bylo vždy velmi nepříjemné. Problém s kvalitou editoru se podařilo vyřešit až s pomocí jiného integrovaného prostředí od společnosti Sigasi – Sigasi Studio XPRT. Toto vývojové prostředí je postavené nad OpenSource projektem Eclipse a je vybaveno integrací s projekty Intel Quartus Prime. Jeho největší výhodou je ovšem schopnost analyzovat v reálném čase rozpracovaný návrh a označovat základní chyby.

Velmi důležitou komponentou *ESM*, které bylo věnováno mnoho času, je ladicí subsystém. Ten měl být původně zcela vlastního návrhu a ke komunikaci používat sériovou linku. Tento postup se záhy ukázal jako nevyhovující. Kromě specifikace protokolu nad sériovou linkou by samozřejmě bylo nutné jej implementovat jak na *FPGA*, tak v rámci ladicího programu. Bylo tedy rozhodnuto najít již existující systém, který se pro potřeby *ESM* upraví. Jako nevhodnější se nakonec ukázal systém navržený pro *RISC-V*, který měl nejen kvalitní dokumentaci a podporu ve známém ladicím programu *OpenOCD*, ale existovala k němu i jednoduchá a přehledná implementace ve *VHDL*, která posloužila jako východisko pro implementaci funkcí potřebných pro *ESM*. Přesto bylo nutné nejprve prostudovat prakticky celou specifikaci tohoto rozhraní, a pak z ní vybrat vhodné prvky pro použití v *ESM*. Během implementace bylo ještě nutné nastudovat část dokumentace samotného procesoru *RISC-V*. Protože důležitou roli v tomto systému hraje také *OpenOCD*, i jeho dokumentace musela být prostudována. Naneštěstí nakonec došlo i na procházení zdrojového kódu, protože nebylo možné nikde jinde zjistit, co přesně vyjadřují výpisy *OpenOCD* zachycující komunikaci s hardwarovou ladicí jednotkou. Nakonec se ale hodiny studia a ladění implementace vyplatily a výsledek této zdlouhavé práce se ihned zúročil při dalším vývoji *ESM*.

Periferie byly přidány jako poslední, jejich implementace nebyla náročná, ale to především proto, že se ji nepodařilo dotáhnout do konce. Podle autorova názoru by měla být co nejdříve od základu přepracována. Za největší problémy aktuální implementace lze považovat její malou integraci se zbytkem mikrokontroléru a žádnou integraci s ladicím subsystémem. Součástí reimplementace by tak mělo být především začlenění registrů periferií do adresového prostoru datové paměti, čímž by zároveň vznikl prostor pro jednoduché zakomponování do ladicího subsystému.

Hotový mikrokontrolér *ESM* ještě potřeboval alespoň primitivní jazyk symbolických instrukcí, aby uživatelé nebyli odkázáni na editor binárních souborů a dokumentaci procesoru. V práci představený jazyk *SEASM* je velmi jednoduchý. To je dáno zejména tím, že jeho překladač není implementován pomocí standardních, robustních nástrojů jako je *Flex*, *Bison* a *Yacc*. Naopak, je napsaný prakticky od nuly v jazyce *Javascript*, který dovoluje celkem rychlé prototypování. Vývoj navíc urychlilo a usnadnilo použití testovacího *frameworku*, kdy testy byly často napsány ještě před implementací funkcionality, a bylo tak velmi jednoduché ověřit, že se vývoj posouvá správným směrem. Vícekrokový proces, který nakonec vede k sestavení programu, vychází z principů, na jakých pracují běžné kompilátory a *linkery*[2] a i proto vede k úspěšnému sestavení spustitelného souboru.

Vytvořit nejen procesor, ale i nástroje kolem něj, byla velká výzva, kterou se nakonec podařilo splnit. A tak může být do výuky zařazena další pomůcka, která možná přitáhne více pozornosti k této odstrčené architektuře.

[2]Sestavovací program; spojuje jednotlivé části programu do spustitelného souboru

Kapitola 5

Závěr

Cíle práce byly návrh a implementace *soft-core* mikrokontroléru s podporou hardwarového ladění a nástrojů pro jeho programování a ladění.

Tyto cíle se podařilo splnit. Mikrokontrolér *EduStackMachine* splňuje stanovená kritéria. Procesor sice nedosahuje takové úrovně optimalizace, jako jiné představené procesory, na druhou stranu je jeho instrukční sada snadno rozšiřitelná a poskytuje dostatek prostoru pro zachování zpětné kompatibility. Další výhodou je vcelku jednoduchá implementace rozdělená do více *VHDL* entit.

K mikrokontroléru byl také vyvinut jazyk symbolických instrukcí *SEAMS* a byl pro něj implementován překladač. *SEASM* podporuje návěští a funkce a konstrukce, které abstrahují část fungování procesoru a usnadňují psaní programů.

EduStackMachine také podporuje hardwarové ladění, které je postaveno na otevřené specifikaci ladicího rozhraní *RISC-V* a používá prověřený ladicí program *OpenOCD*.

Do budoucna by bylo vhodné zvážit rozšířit projekt *ESM* o simulátor, ten by mohl sloužit místo fyzických přípravků pro domácí úlohy. Dalším užitečným doplňkem by mohlo být grafické rozhraní pro ladění procesoru, které by zobrazovalo hodnoty vyčtené z fyzického přípravku.

Přestože vývoj takto rozsáhlého projektu byl náročný a dlouhý, je jeho výsledkem zásobníkový mikrokontrolér doplněný o potřebné nástroje, který bude možné, i přes některé drobné vady, nasadit do výuky.

Bibliografie

[1] *ATmega328PB*. Microchip Technology Inc. 2018. ISBN: 978-1-5224-2681-3.

[2] James Bowman. „J1: a small Forth CPU Core for FPGAs“. In: *26th EuroForth Conference*. 2010, s. 43–46. URL: http://www.complang.tuwien.ac.at/anton/euroforth/ef10/papers/bowman.pdf.

[3] Jakub Cabal. *Simple UART for FPGA*. 2015-2021. URL: https://github.com/jakubcabal/uart-for-fpga.

[4] *DE10-Lite Board*. TerasIC. 2020. URL: https://web.archive.org/web/20220727150804/https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English%5C&No=1021 (cit. 03. 09. 2022).

[5] *Exploring the J1 Instruction Set and Architecture using SIMPL*. 2017. URL: https://nanode0000.wordpress.com/2017/04/08/exploring-the-j1-instruction-set-and-architecture/ (cit. 23. 09. 2022).

[6] Øyvind Harboe, Álvaro Lopes a bert-lange. *The Zylin ZPU*. Dub. 2015. URL: https://github.com/zylin/zpu.

[7] Richard Howe. *Forth SoC written in VHDL*. 2013-2019. URL: https://github.com/howerj/forth-cpu.

[8] Chair of VLSI Design, Diagnostics and Architecture. *PoC - Pile of Cores*. Technische Universität Dresden. 2016. URL: https://github.com/VLSI-EDA/PoC (cit. 28. 10. 2016).

[9] *Intel® MAX® 10 FPGA Device Datasheet*. 683794. Version: 2022.10.31. Intel Corporation. Říj. 2022.

[10] Philip Koopman Jr. *Stack computers: The new wave*. 1. vyd. Pittsburgh, USA: Ellis Horwood, 1989. ISBN: 9780745804187.

[11] Hoda Aghaei Khouzani a Chengmo Yang. „A DWM-Based Stack Architecture Implementation for Energy Harvesting Systems“. In: *ACM Trans. Embed. Comput. Syst.* 16.5s (zář. 2017). ISSN: 1539-9087. DOI: 10.1145/3126543. URL: https://doi.org/10.1145/3126543.

[12] Donald Ervin Knuth. *The art of computer programming*. 3rd ed. Reading, Mass.: Addison-Wesley, 1997. ISBN: 02-018-9683-4.

[13] Ryosuke Nakamura, Yasuaki Ito a Koji Nakano. „TinyCSE: Tiny Computer System for Education“. In: *2013 First International Symposium on Computing and Networking*. 2013, s. 639–641. DOI: 10.1109/CANDAR.2013.117.

[14] Koji Nakano a Yasuaki Ito. „Processor, Assembler, and Compiler Design Education Using an FPGA“. In: *2008 14th IEEE International Conference on Parallel and Distributed Systems*. 2008, s. 723–728. DOI: 10.1109/ICPADS.2008.71.

[15] Tim Newsome a Megan Wachs. *RISC-V External Debug Support*. Version: 0.13.2. RISC-V International. Dub. 2019.

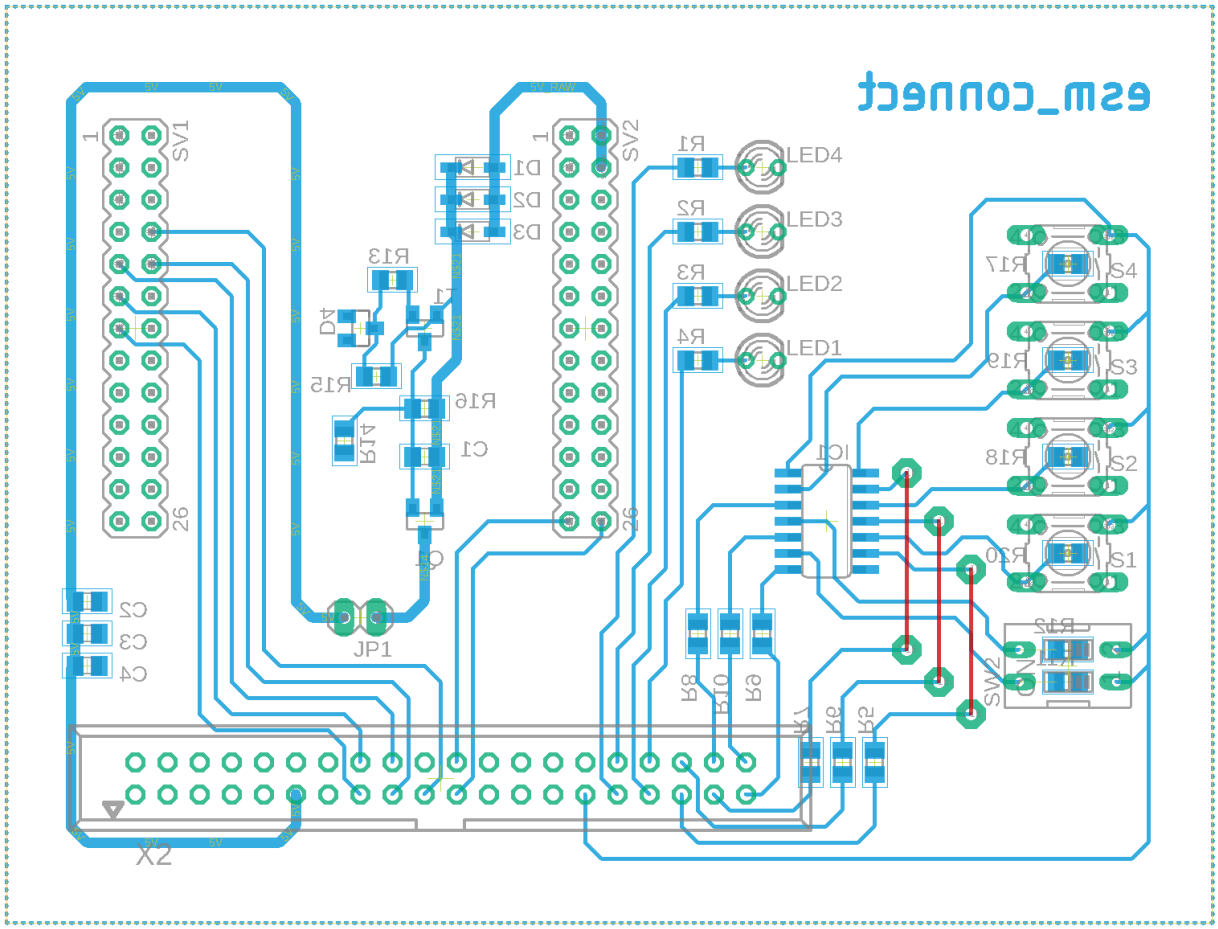
[16] Stephan Nolting a All the Awesome Contributors. *The NEORV32 RISC-V Processor*. Zář. 2022. DOI: 10.5281/zenodo.7030070. URL: https://github.com/stnolting/neorv32.

[17] Marc Reichenbach, Benjamin Pfundt a Dietmar Fey. „Designing and Manufacturing of Real Embedded Multi-Core CPUs: A Holistic Teaching Approach in Computer Architecture“. In: květ. 2014. DOI: 10.1109/EWME.2014.6877428.

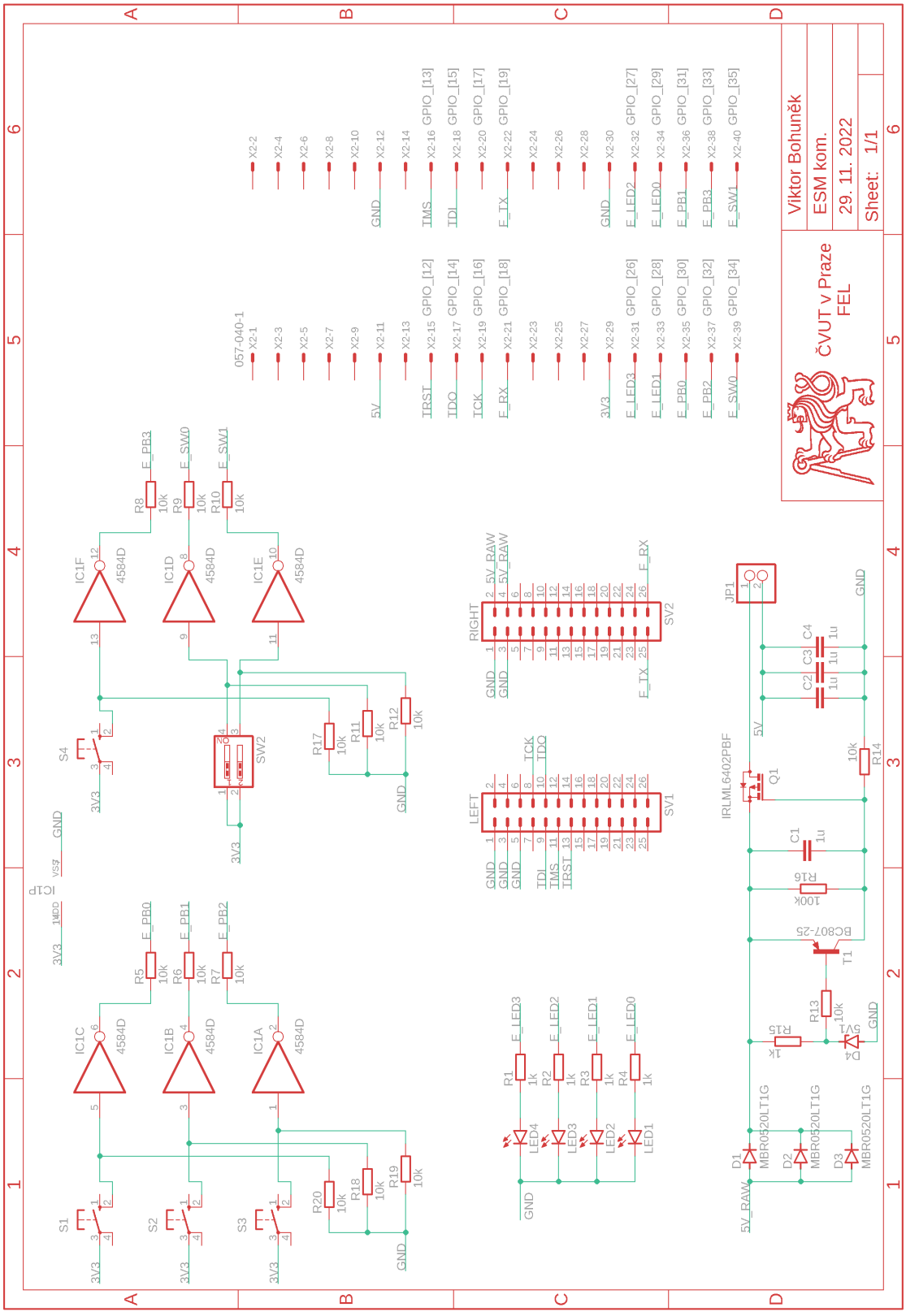
[18] Fang Su et al. „Nonvolatile processors: Why is it trending?“ In: *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. 2017, s. 966–971. DOI: 10.23919/DATE.2017.7927131.

Příloha A

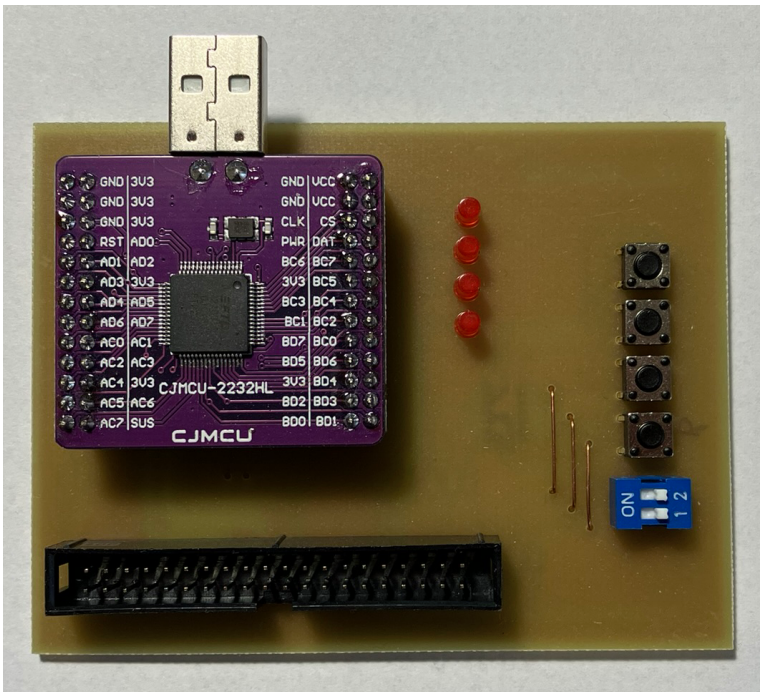
Obrázky



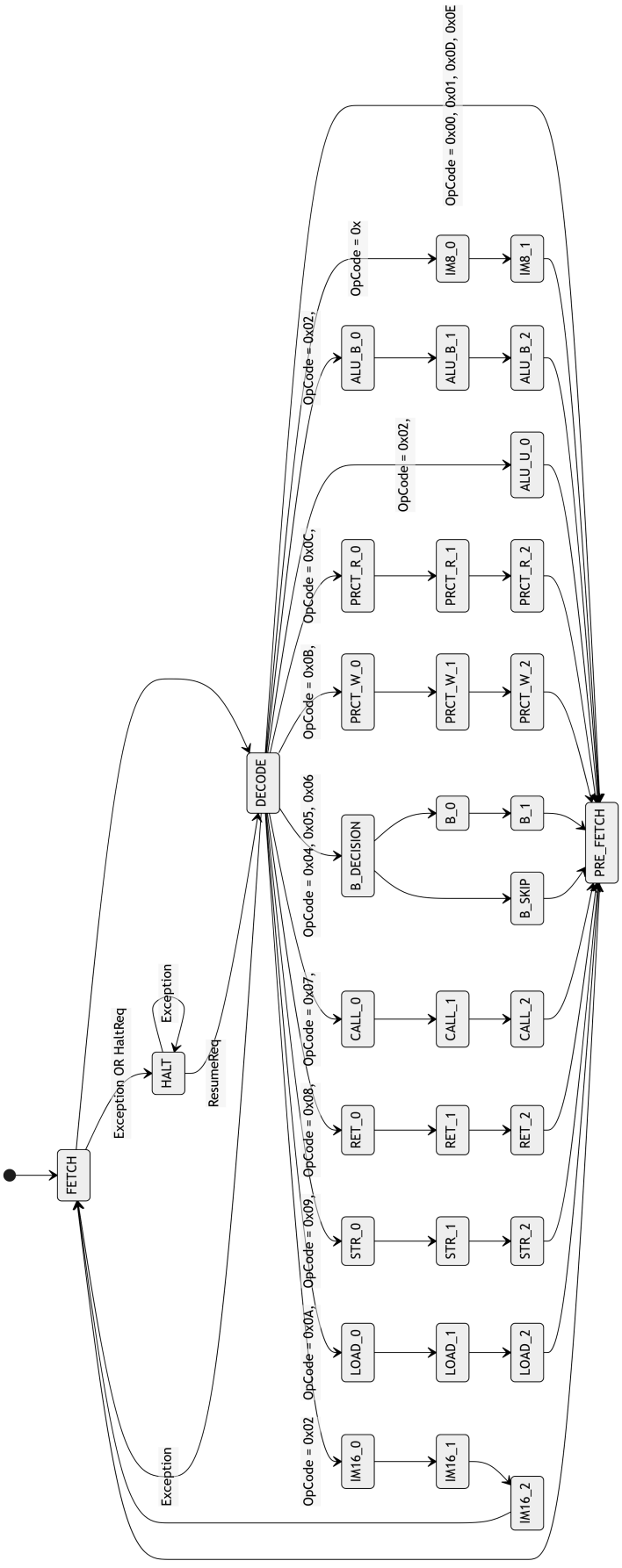
Obrázek A.1: Motiv plošného spoje pro komunikační desku, bez rozlité mědi; Spoje z *TOP* vrstvy jsou na prototypu realizovány drátovými propojkami



Obrázek A.2: Schéma komunikační desky



Obrázek A.3: Fotografie komunikační desky



Obrázek A.4: Stavový automat řadiče *ESM*

Příloha B

Dokumentace *EduStackMachine*

B.1 Přehled

EduStackMachine je 16bitový mikrokontrolér postavený na zásobníkovém procesoru harvardské architektury určený pro použití ve výuce. Cílem tohoto dokumentu je představit jak mikrokontrolér funguje na úrovni instrukční sady a jak používat jeho vestavěnou ladicí jednotku.

- 16bitová zásobníková architektura s variabilní délkou instrukce
  - little-endian
  - 23 instrukcí
  - průměrně je jedna instrukce vykonána za 5 strojových cyklů
  - hardwarový datový zásobník s hloubkou 64 16bitových slov
  - hardwarový zásobník návratových adres s hloubkou 16 16bitových slov
  - průměrně 0.2 MIPS při 1 MHz
  - 8 KiB programové paměti SRAM
  - 16 KiB paměti RAM
- periferie
  - 16 GPIO pinů s rozložením kompatibilní s Arduino UNO rozhraním
  - UART s přenosovou rychlostí 9600 baud
  - 6 sedmisegmentových displejů bez multiplexování, každý segment je adresovatelný zvlášť
  - 10 přepínáčů
  - 2 spínače
- podpora hardwarového ladění pomocí OpenOCD přes JTAG
  - přístup ke všem pamětem
  - krokování programu
  - možnost uložit více programů a přepínat mezi nimi

B.2 Použité pojmy

ESM – zkrácený název mikrokontroléru, pro který je psána dokumentace, plný název *EduStackMachine*

MSB – most Significant Bit - bit v rámci vyššího celku o N bitech; na první pozici zleva, s číslem $N-1$ který má váhu 2^N

LSB – least Significant Bit - bit v rámci vyššího celku o N bitech; na poslední pozici zleva, vždy s číslem 0 a váhou 1

byte – 8 bitů

slovo – 16 bitů, v pořadí 15-0, kde 15 je MSB a 0 je LSB

horní byte – bity 15-8 v rámci `slova`

spodní byte – bity 7-0 v rámci `slova`

PMEM – programová paměť

DMEM – datová paměť

DS – datový zásobník

RS – zásobník návratových adres

TOPD0 – vrchní `slovo` v `DS`

TOPD1 – `Slovo` na pozici pod `TOPD0` v `DS`

TOPR – vrchní `slovo` v `RS`

OpCode – `Byte` s hodnotou, která identifikuje konkrétní instrukci

ROM – read only memory = paměť pouze pro čtení

RAM – random access memory = paměť s náhodným přístupem s možností čtení i zápisu

program – sekvence instrukcí uložená v `PMEM`

OpenOCD – open On-Chip Debbuger – ladicí program, který komunikuje s ladicí jednotkou

KiB – kibibyte = 1024 bytů

R – přístup pro čtení

R/W – přístup pro čtení a zápis

W – přístup pouze pro zápis, při čtení vrací 0

WARL – je možné zapsat libovolnou hodnotu, ale čtena bude pouze platná hodnota

B.3 Použité symboly

Symbol	Význam
==	test rovnosti obou stran (0 == 0 je pravda)
!=	test nerovnosti obou stran (0 != 1 je pravda)
=	přiřazení hodnoty napravo do prvku vlevo
+	aritmetické sčítání
-	aritmetické odečítání
AND	bitově orientovaná funkce logického součinu
OR	bitově orientovaná funkce logického součtu
XOR	bitově orientovaná funkce exkluzivního logického součtu
X << n	bitový posun čísla X vlevo o n bitů
X >> n	bitový posun čísla X vpravo o n bitů

B.4 Paměť

Protože `ESM` je harvardské architektury, je vybaven dvěma oddělenými pamětmi – datovou (`DMEM`) a programovou (`PMEM`).

DMEM

Datová paměť je organizována jako 16 × 8k a má kapacitu 16 KiB. Je přístupná pouze po `slovech`. Z pohledu `ESM` se jedná o `RAM`, pro manipulaci se používají instrukce `STORE` a `LOAD`.

Paměť je dostupná skrz ladicí jednotku jako 8 × 16k. Je tedy adresovatelná po jednotlivých `bytech`.

PMEM

Datová paměť je organizována jako 8 × 8k a má kapacitu 8 KiB. Je přístupná pouze po `bytech`. Z pohledu `ESM` se jedná o `ROM`, není tedy možné pomocí `programu` měnit její obsah.

Paměť je dostupná skrz ladicí jednotku jako 8 × 8k. Kromě přístupu po `bytech` je k dispozici také přístup po `slovech`, ten je ale určen zejména pro dávkové čtení a zápis z `OpenOCD`.

B.5 Zásobníky

DS

- datový zásobník
- přístup pro `program`: `R/W`
- přístup z ladicí jednotky: `R/W`
- 64 `slov`
- výjimka při podtečení, přetečení viz `EXC`

`DS` u `ESM` nahrazuje registry, jeho dva vrchní prvky `TOPD0` a `TOPD1` vstupují jako operandy do většiny instrukcí. Před vykonáním každé instrukce se kontroluje, že nehrozí přetečení či podtečení zásobníku, pokud by takový stav měl nastat, `ESM` zastaví vykonávání programu a uloží do registru `EXC` hodnotu odpovídající příslušné výjimce.

RS

- datový zásobník
- přístup pro `program`: `R/W`
- přístup z ladicí jednotky: `R/W`
- 16 `slov`
- výjimka při podtečení, přetečení viz `EXC`

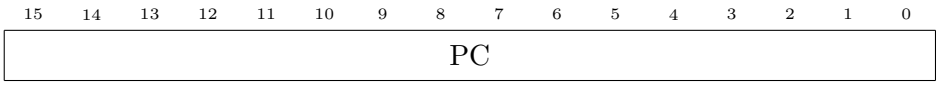
Zásobník návratových adres funguje podobně jako zásobník u procesorů s registry. V případě `ESM` se na něj ale ukládají jen návratové adresy, nikdy na něj nejsou ukládána data. Se zásobníkem operují pouze instrukce `CALL` a `RET`, před jejich vykonáním se kontroluje, jestli nehrozí přetečení/podtečení a pokud ano, tak `ESM` zastaví vykonávání programu a uloží do registru `EXC` hodnotu odpovídající příslušné výjimce.

B.6 Registry

PC

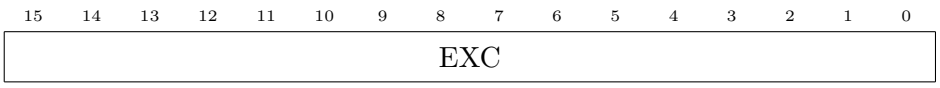
- program Counter
- přístup pro `program`: `W`
- přístup z ladicí jednotky: `R/W`

Do registru zapisují funkce `CALL`, `RET`, `ZB`, `NZB` a `UB`



EXC

- exception
- přístup pro `program`: `--`
- přístup z ladicí jednotky: `R`



Hodnota	Význam
`0x0000`	žádná výjimka
`0x0001`	přetečení `DS`
`0x0002`	podtečení `DS`
`0x0003`	přetečení `RS`
`0x0004`	podtečení `RS`
`0x0005`	dekódována nedefinovaná instrukce

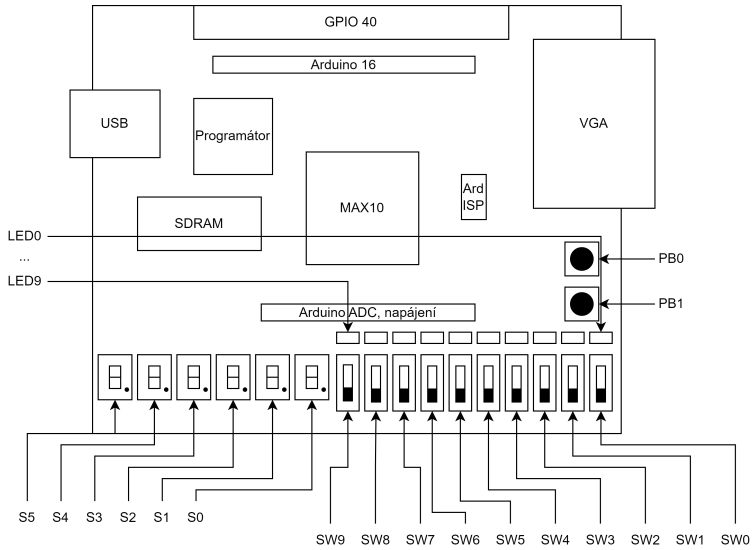
Tabulka B.1: Význam hodnot v registru `EXC`

B.7 Periferie

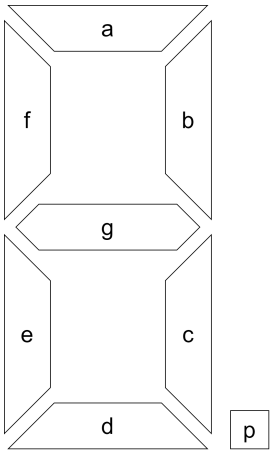
O přístup k periferiím se stará řadič periferií, ten obsahuje sadu registrů, popsaných níže, které umožňují interagovat s periferiemi.

B.7.1 Registry periferií

V této sekci jsou popsány jednotlivé bity všech registrů řadiče periferií. Popisky v bitech odpovídají názvům z obr. B.1 a v případě sedmisegmentových displejů také obr. B.2.



Obrázek B.1: Přehled periferií na desc *DE10-Lite*



Obrázek B.2: Pojmenování segmentů sedmisegmentového displeje

PARDI

- arduino GPIO - input
- přístup pro `program`: `WARL`
- přístup z ladicí jednotky: `--`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
A15	A14	A13	A12	A11	A10	A09	A08	A07	A06	A05	A04	A03	A02	A01	A00

PARDO

- arduino GPIO - output
- přístup pro `program`: `R/W`
- přístup z ladicí jednotky: `--`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
A15	A14	A13	A12	A11	A10	A09	A08	A07	A06	A05	A04	A03	A02	A01	A00

PARDE

- arduino GPIO - enable
- přístup pro `program`: `R/W`
- přístup z ladicí jednotky: `--`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
A15	A14	A13	A12	A11	A10	A09	A08	A07	A06	A05	A04	A03	A02	A01	A00

A15-A00 – 0 znamená, že pin je nastaven jako vstupní (je v režimu vysoké impedance); 1 znamená, že je pin nastaven jako výstupní

PLEDO

- LED – output
- přístup pro `program`: `R/W`
- přístup z ladicí jednotky: `--`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0						L09	L08	L07	L06	L05	L04	L03	L02	L01	L00

PSEG0O

- 7-segment display 0 – output
- přístup pro `program`: `R/W`
- přístup z ladicí jednotky: `--`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
S1a	S1b	S1c	S1d	S1e	S1f	S1g	S1p	S0a	S0b	S0c	S0d	S0e	S0f	S0g	S0p

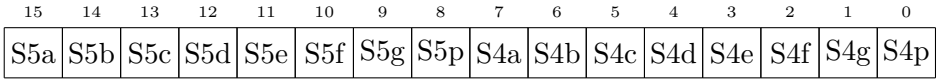
PSEG1O

- 7-segment display 1 – output
- přístup pro `program`: `R/W`
- přístup z ladicí jednotky: `--`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
S3a	S3b	S3c	S3d	S3e	S3f	S3g	S3p	S2a	S2b	S2c	S2d	S2e	S2f	S2g	S2p

PSEG2O

- 7-segment display 2 – output
- přístup pro `program`: `R/W`
- přístup z ladicí jednotky: `--`

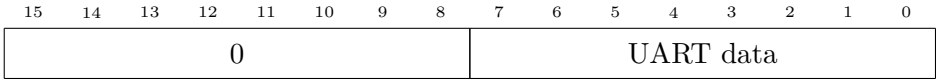


PSWPBI

- switches and push-buttons – input
- přístup pro `program`: `WARL`
- přístup z ladicí jednotky: `--`

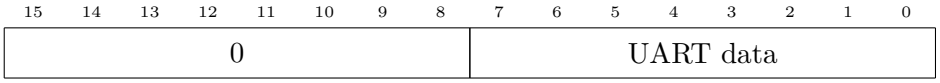
PUARTI

- UART – input
- přístup pro `program`: `WARL`
- přístup z ladicí jednotky: `--`



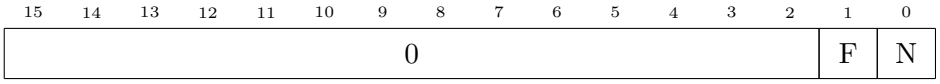
PUARTO

- UART – output
- přístup pro `program`: `W`
- přístup z ladicí jednotky: `--`



PUARTS

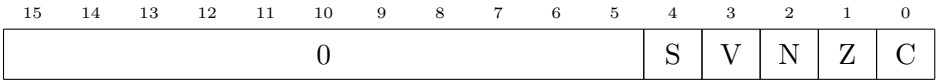
- UART – status
- přístup pro `program`: `WARL`
- přístup z ladicí jednotky: `--`



F – Full; pokud je bit 1, je plná odesílací fronta

N – New; pokud je bit 1, čeká ve frontě alespoň jeden byte na přečtení

B.8 ALU flags



Obrázek B.3: Příznaky poskytované ALU

C Carry – přetečení, příznak je nastaven na 1, pokud výsledek matematické operace překročil rozsah 16 bitů; při operaci bitového posunu je v něm uložen bit, který je vysunut mimo rozsah

Z Zero – poslední výsledek libovolné operace byl 0

N Negative – poslední výsledek matematické operace měl bit 15 nastaven na 1, tedy ve dvojkovém doplňku se jedná o záporné číslo

V Two's complement overflow – při poslední matematické operaci došlo k přetečení dvojkového doplňku

S Sign – určen jako exkluzivní logický součet (XOR) příznaku `N` a `V`, vyjadřuje jaké znaménko by měl mít výsledek poslední matematické operace

B.8.1 Podmínky přetečení dvojkového doplňku

Při sčítání může dojít k přetečení jen pokud mají sčítance stejné **znaménko** a výsledek má znaménko **opačné**, tedy $(-\text{TOPD1}) = (+\text{TOPD1}) + (+\text{TOPD1})$ a $(+\text{TOPD0}) = (-\text{TOPD1}) + (-\text{TOPD0})$, kde znaménka v závorkách vyjadřují stav bitu 15, tj. „+“ pokud je 0 a „-“ pokud je 1.

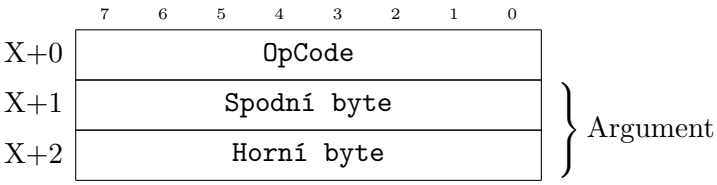
Při odečítání může dojít k přetečení jen pokud mají sčítance **různá** znaménka a výsledek má znaménko **stejné jako menšitel**, tedy $(-\text{TOPD1}) = (+\text{TOPD1}) - (-\text{TOPD1})$ a $(+\text{TOPD0}) = (-\text{TOPD1}) - (+\text{TOPD0})$, kde znaménka v závorkách vyjadřují stav bitu 15, tj. „+“ pokud je 0 a „-“ pokud je 1.

B.9 Instrukce

V této sekci jsou popsány všechny instrukce z instrukční sady `ESM`. U každé instrukce jsou uvedeny základní údaje a krátký popis, případně další informace o jejich podstatných vlastnostech.

V aktuální verzi instrukční sady jsou instrukce dlouhé 1–3 byty, podle toho jaký argument potřebují. Jednotlivé instrukce (a tedy i byty) jsou umístěny hned za sebou, protože programová paměť je přístupná po bytech. Není tak potřeba řešit zarovnávání, které by nejen komplikovalo tvorbu binárních souborů, ale také by snižovalo hustotu instrukcí uložených v paměti.

Uložení instrukcí a jejich argumentů v paměti je znázorněno pomocí diagramu, jehož vzor je na obr. B.4. V diagramu rostou adresy programové paměti směrem dolů. Základní adresa, označená jako X, je ta, na které je uložen `OpCode`. Pokud instrukce vyžaduje argument, je uložen v bytech za `OpCode`.



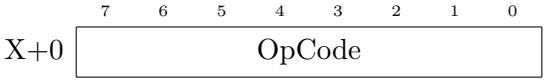
Obrázek B.4: Diagram uložené instrukce v paměti, X v nalevo označuje adresu, na které je umístěn `OpCode`

NOP

Název: No operation

OpCode: `0x00`

- instrukce nevykonává žádnou operaci, lze ji využít pro realizaci čekacích stavů.
- instrukce je vykonána během 3 cyklů, při taktovací frekvenci 1 MHz to zpoždění znamená 30 µs.

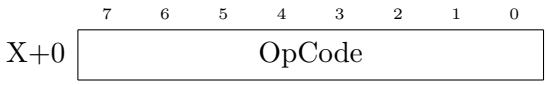


BP

Název: Break point

OpCode: `0x01`

- tato instrukce bez aktivovaného ladicího režimu (viz `Ladění programu`) funguje jako `NOP`.
- po připojení OpenOCD k ladicí jednotce je automaticky aktivován ladicí režim a procesor po dekódování instrukce `BP` inkrementuje `PC` a zastaví vykonávání programu. Během ladění je tedy v `PC` k dispozici adresa **příští** vykonávané instrukce.

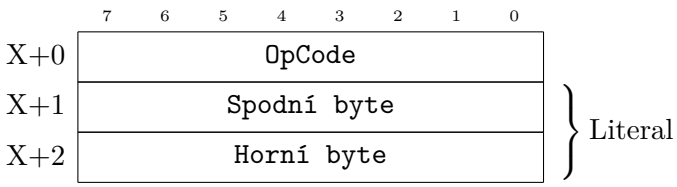


IM16

Název: 16bit integer literal

OpCode: `0x02`

- instrukce načte z `PMEM` 16 bitů z následujících adres po adrese svého `OpCode` a vloží je do `TOPDO`

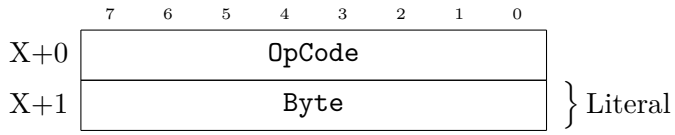


IM8

Název: 8bit integer literal

OpCode: `0x03`

- instrukce načte z PMEM 8 bitů z následující adresy po adrese svého `OpCode` a vloží je do `TOPD0`
- tato instrukce je vykonána o 1 cyklus rychleji než IM16
- `ALU ESM` nepodporuje výpočty v dvojkovém doplňku pro 8 bitová čísla, tedy záporná čísla lze zadávat jen pomocí IM16

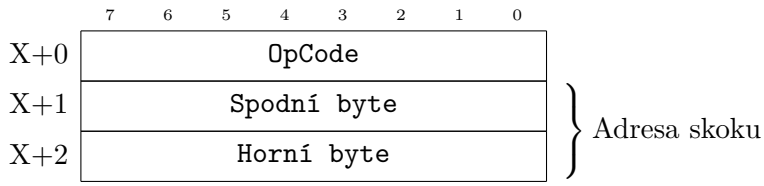


ZB

Název: Zero branch

OpCode: `0x04`

- instrukce podmíněného skoku
- podmínkou pro skočení je, že `TOPD0` == 0
- při skoku zapíše hodnotu z bytů následujících po `OpCode` do PC

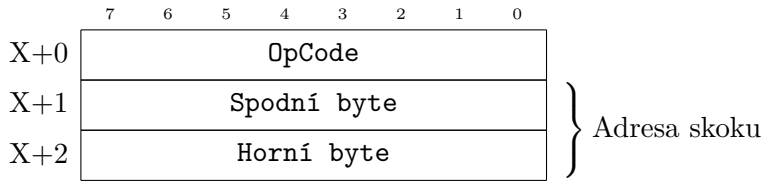


NZB

Název: Non-zero branch

OpCode: `0x05`

- instrukce podmíněného skoku
- podmínkou pro skočení je, že `TOPD0` != 0
- při skoku zapíše hodnotu z bytů následujících po `OpCode` do PC

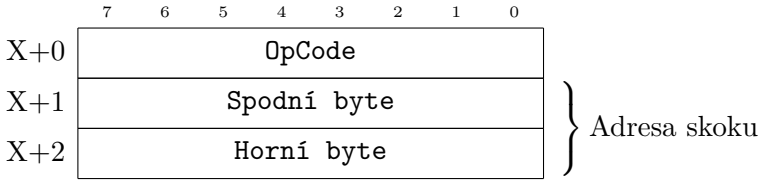


UB

Název: Unconditional branch

OpCode: `0x06`

- nepodmíněný skok
- instrukci je vhodné použít na konci každého programu a vytvořit tak nekonečnou smyčku
- při vykonání zapíše hodnotu z bytů následujících po `OpCode` do `PC`

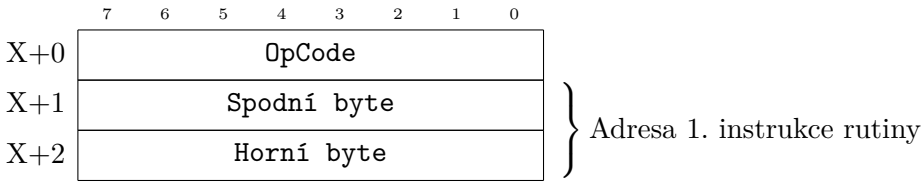


CALL

Název: Call

OpCode: `0x07`

- při vykonání nejprve uloží aktuální hodnotu `PC` do `TOPR` a potom do `PC` zapíše hodnotu z bytů následujících po `OpCode`

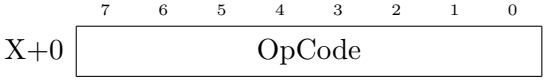


RET

Název: Return

OpCode: `0x08`

- při vykonání uloží hodnotu z `TOPR` do `PC`



STORE

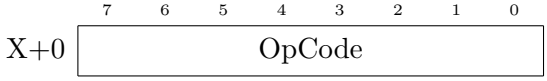
Název: Store

OpCode: `0x09`

Adresa z: `TOPD1`

Hodnota z: `TOPD0`

- instrukce uloží hodnotu v `TOPD0` na adresu v `DMEM` uvedenou v `TOPD1`
- během vykonávání jsou oba parametry odstraněny z `DS`



LOAD

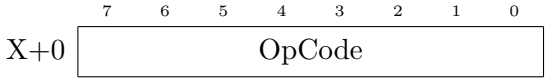
Název: Load

OpCode: `0x0A`

Adresa z: `TOPD0`

Hodnota do: `TOPD0`

- instrukce uloží hodnotu v `DMEM` na adrese uvedené v `TOPD0` do `TOPD0`, tedy hodnota z paměti přepíše původně uvedenou adresu v `TOPD0`



PERIFWR

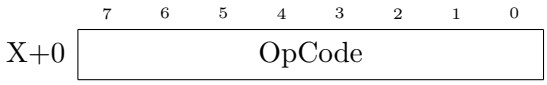
Název: Peripherals write

OpCode: `0x0B`

Adresa z: `TOPD1`

Hodnota z: `TOPD0`

- instrukce uloží hodnotu v `TOPD0` do registru na adrese v řadiči periferií uvedenou v `TOPD1`
- během vykonávání jsou oba parametry odstraněny z `DS`



PERIFRD

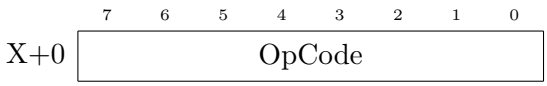
Název: Peripherals read

OpCode: `0x0C`

Adresa z: `TOPD0`

Hodnota do: `TOPD0`

- instrukce uloží z registru na adrese uvedené v `TOPD0` v řadiči periferií do `TOPD0`, hodnota tedy přepíše původně uvedenou adresu v `TOPD0`

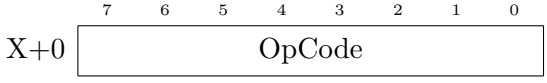


DIS

Název: Discard

OpCode: `0x0D`

- instrukce smaže `TOPD0`

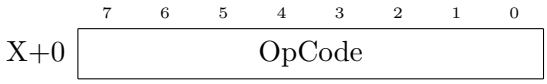


DUP

Název: Duplicate

OpCode: `0x0E`

- instrukce duplikuje `TOPD0`, tedy hodnota v `TOPD0` je znovu vložena do `DS`

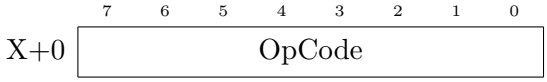


ADD

Název: Addition

OpCode: `0x20`

Operace: `TOPD1` = `TOPD1` + `TOPD0`; `TOPD0` = `ALU flags`

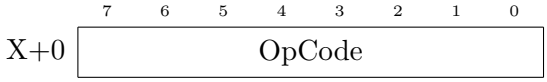


SUB

Název: Subtraction

OpCode: `0x21`

Operace: `TOPD1` = `TOPD1` - `TOPD0`; `TOPD0` = `ALU flags`

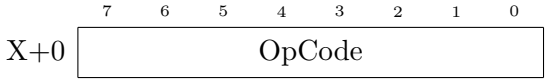


LSHIFT

Název: Bitwise left-shift

OpCode: `0x22`

Operace: `TOPD1` = `TOPD0` << 1; `TOPD0` = `ALU flags`

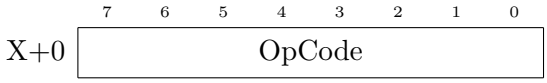


RSHIFT

Název: Bitwise right-shift

OpCode: `0x23`

Operace: `TOPD1` = `TOPD0` >> 1; `TOPD0` = `ALU flags`

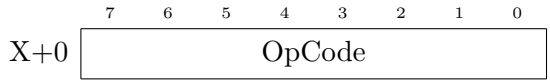


NOT

Název: Bitwise NOT

OpCode: `0x28`

Operace: `TOPD1` = NOT `TOPD0`; `TOPD0` = `ALU flags`

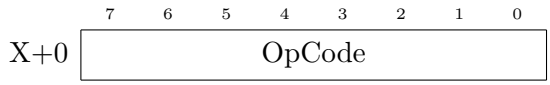


AND

Název: Bitwise AND

OpCode: `0x29`

Operace: `TOPD1` = `TOPD1` AND `TOPD0`; `TOPD0` = `ALU flags`

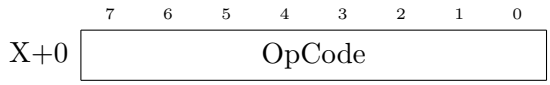


OR

Název: Bitwise OR

OpCode: `0x2A`

Operace: `TOPD1` = `TOPD1` OR `TOPD0`; `TOPD0` = `ALU flags`

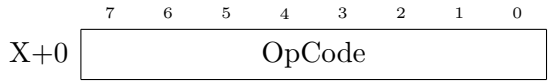


XOR

Název: Bitwise XOR

OpCode: `0x2B`

Operace: `TOPD1` = `TOPD1` XOR `TOPD0`; `TOPD0` = `ALU flags`



B.10 Ladění programu

Ladění programu je řízeno z `OpenOCD`, ke kterému je potřeba se připojit přes jeho `telnet` server. Aby bylo `ESM` detekováno, musí se `OpenOCD` spustit s konfiguračním skriptem uloženým v `esm_openocd/esm_ftdi.cfg`.

Po spuštění `OpenOCD` naskenuje `JTAG` řetěz a inicializuje v něm zapojenou ladicí jednotku v `ESM`. Na konci procesu inicializace je `ESM` zastaveno, aby bylo možné rovnou začít zadávat příkazy. V konfiguračním souboru jsou přidány výpisy pro zastavení programu (`halted`), jeho obnovení (`resumed`) a krokování (`stepped`), aby byl uživatel informován o stavu `ESM`.

`ESM` se ovládá jak pomocí vestavěných příkazů `OpenOCD`, tak příkazů přidaných v inicializačním skriptu:

halt – `OpenOCD` požádá `ESM` o zastavení programu, měla by následovat událost `halted`

resume (addr) – OpenOCD požádá ESM o obnovení chodu programu, měla by následovat událost resumed; nepovinný argument addr dovoluje obnovit chod programu na jiné adrese v PMEM než na jaké byl zastaven

load_image soubor adresa – Příkaz pro nahrání binárního souboru ze složky, kde je spuštěno OpenOCD na zadanou adresu

dump_image soubor adresa délka – Příkaz pro zkopírování obsahu paměti do binárního souboru do složky, kde je spuštěno OpenOCD z adresy o zadané délce v bytech (pozor – u DMEM, DS a RS je tato délka dvojnásobná, než z pohledu ESM)

esm_enb on|off – Povolí nebo zakáže zastavení programu na instrukci BP

esm_pc – Vypíše stav registru PC

esm_aludf (adresa) – Vypíše byte v TOPD0 s nadepsanými písmenky pro ALU flags, pokud je zadána adresa, tak se přečte slovo na dané adrese v DS

reset (halt) – Příkaz restartuje ESM; pokud je přidán ještě argument halt, nedojde po resetu ke spuštění programu, ESM bude zastaveno před první instrukcí

esm_haltr – Vypíše důvod zastavení programu

esm_dsp – Ukazatel do DS používaný logikou ESM, je pouze pro čtení; poskytuje informaci o zaplněnosti

esm_dsr adresa – Čtení z DS, adresa je z rozsahu $\langle 0, 63 \rangle$

esm_dsw adresa data – Zápis dat do DS na adresu z rozsahu $\langle 0, 63 \rangle$

esm_rsp – Ukazatel do RS používaný logikou ESM, je pouze pro čtení; poskytuje informaci o zaplněnosti

esm_rsr adresa – Čtení z RS, adresa je z rozsahu $\langle 0, 15 \rangle$

esm_rsw adresa data – Zápis dat do RS na adresu z rozsahu $\langle 0, 15 \rangle$

mdhx – Vylepšení funkce mdh o prefix 0x u hodnoty, výstup je díky tomu přehlednější

esm_topd0 – Vypíše hodnotu z TOPD0, nelze do ní zapisovat

esm_topd1 – Vypíše hodnotu z TOPD1, nelze do ní zapisovat

esm_topr – Vypíše hodnotu z TOPR, nelze do ní zapisovat

Při práci s pamětí přes v OpenOCD se používá následující mapování adres na fyzické paměti uvnitř ESM:

Adresa počáteční	koncová	Paměť	Poznámka
0x00000000	0x00001FFF	Programová paměť	
0x00002000	0x0000FFFF	–	
0x00010000	0x00013FFF	Datová paměť	Z pohledu ladicího subsystému je uspořádána jako 8 × 16k, tedy má 14 adresových vodičů místo 13 z pohledu *ESM*
0x00014000	0x0001FFFF	–	
0x00020000	0x00020080	Datový zásobník	Adresován po bytech, stejně jako datová paměť
0x00020081	0x0002FFFF	–	
0x00030000	0x00030020	Zásobník návratových adres	Adresován po bytech, stejně jako datová paměť
0x00030021	0xFFFFFFFF	–	

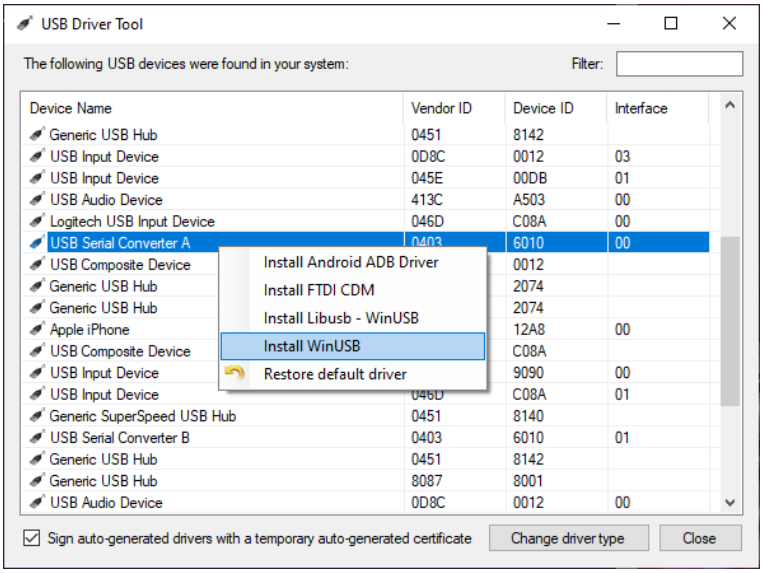
Příloha C

Návod na zprovoznění *OpenOCD*

C.1 OpenOCD – OS Windows 10

Nejprve je potřeba nainstalovat správné ovladače pro převodník FT2232HL na komunikační desce (3.8):

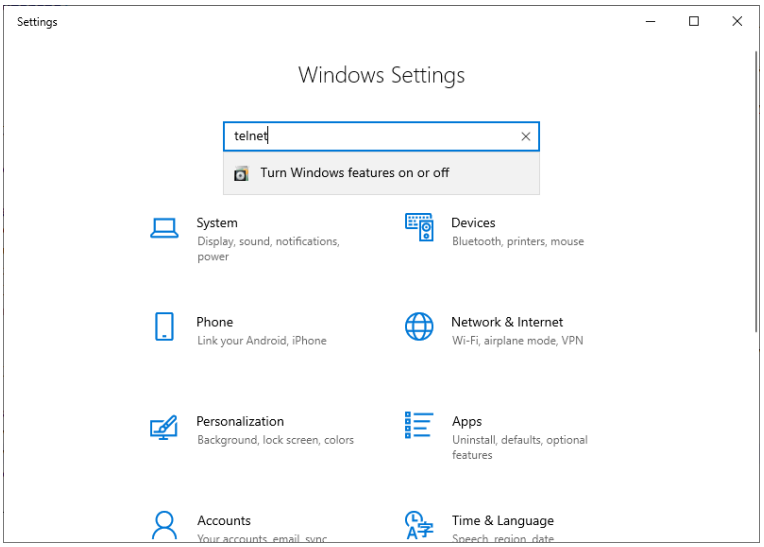
1. ze stránek `https://visualgdb.com/UsbDriverTool` stáhneme samorozbalitelný archiv `UsbDriverTool.exe`,
2. stažený archiv rozbalíme do vhodného adresáře,
3. připojíme komunikační desku k PC (`UsbDriverTool.exe` hledá jen mezi již připojenými zařízeními),
4. spustíme program `UsbDriverTool.exe`,
5. stejně jako na obr. C.1 najdeme zařízení pojmenované „USB Serial Converter A“, s `Vendor ID = 0403` a `Device ID = 6010`, klikneme na něj pravým tlačítkem a z nabídky vybereme „Install WinUSB“,
6. tím je instalace ovladačů ukončena.



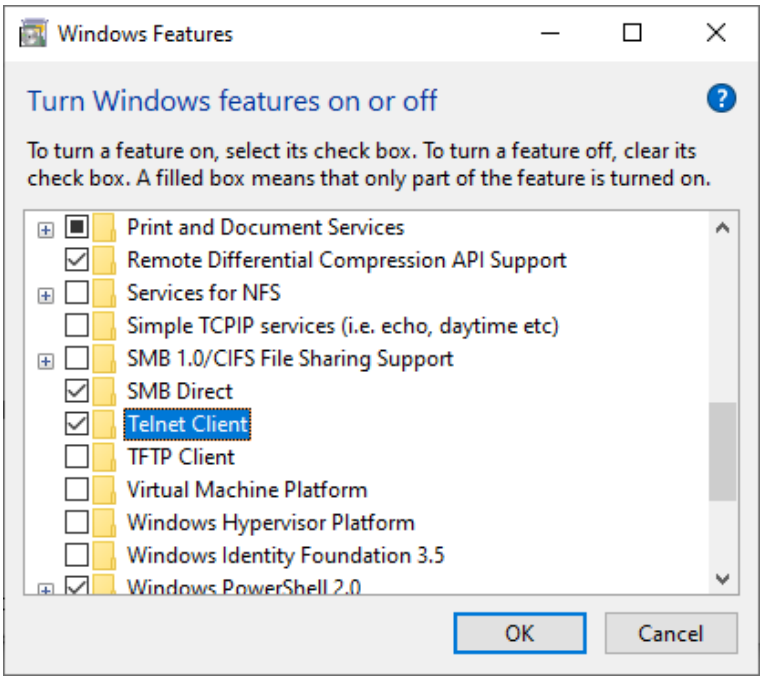
Obrázek C.1: Program `UsbDriverTool.exe`

Pro připojení k *OpenOCD* je možné využít několik protokolů, pro přímou interakci s *OpenOCD* je však určen `telnet`. OS Windows 10 v dnešní době nemá vestavěného klienta standardně zapnutého a je potřeba jej povolit:

1. otevřeme nabídku nastavení,
2. do vyhledávače zadáme „telnet“, měli bychom vidět to samé, co na obr. C.2,
3. po kliknutí na nápovědu vyhledávání se otevře okno s nabídkou doplňkových funkcí OS Windows, je v ní potřeba najít a zaškrtnou „Telnet client“, viz obr. C.3,
4. po uzavření okna tlačítkem OK se zobrazí dialogové okno informující o průběhu instalace,
5. pokud vše proběhlo v pořádku, `telnet` klient je aktivní je možné jej použít z příkazové řádky pro připojení k *OpenOCD*.



Obrázek C.2: Vyhledávání v nastavení OS Windows 10



Obrázek C.3: Vyhledávání v nastavení OS Windows 10

Druhou možností je použít aplikaci třetí strany, např. Putty viz `https://www.putty.org/`.

Pro připojení k ladicímu subsystému *ESM* je potřeba ladicí program *OpenOCD* minimálně ve verzi 0.11.0, ale nejlépe v poslední dostupné verzi, v době psaní manuálu 0.12.0-rc3.

1. *OpenOCD* pro Windows stáhneme z mirroru oficiálního repozitáře na adrese `https://github.com/openocd-org/openocd/releases`,
2. stažený archiv rozbalíme do vhodného adresáře,
3. otevřeme příkazovou řádku (např. z nabídky Start po vyhledání cmd.exe),
4. přejdeme do adresáře `esm_openocd`,
5. spustíme *OpenOCD* příkazem v příkazové řádce `.\cesta\k\adresáři\openocd -f esm_ftdi.cfg`,
6. po inicializaci vypíše *OpenOCD* do příkazové řádky, že je připraveno přijímat připojení přes telnet server na adrese 127.0.0.1:4444, stačí se tedy pomocí zvoleného klienta připojit,
7. v případě vestavěného klienta v OS Windows je potřeba otevřít další příkazovou řádku a v ní teprve spustit příkaze `telnet` bez argumentů,
8. po spuštění klienta je potřeba zadat příkaz k připojení `o 127.0.0.1 4444`.

Popis práce s *OpenOCD* je součástí Dokumentace *EduStackMachine*.

Příloha D

Manuál překladače *SEASM*

SEASM je překladač pro jednoduchý stejnojmenný jazyk symbolických adres navržený pro instrukční sadu mikrokontroléru *ESM*. Překladač produkuje ze vstupního souboru binární spustitelný kód pro `ESM`.

Podrobnější popis instrukcí, na které je kód překladačem *SEAMS* přeložen, je v sekci Dokumentace *EduStackMachine*.

D.1 Spuštění překladače

Překladač *SEASM* je napsaný v *Javascriptu*, a pro spuštění vyžaduje nainstalované prostředí *Node.js* ve verzi 18. Pro nainstalování balíčků, které překladač využívá, je ještě nutné mít nainstalovaný *yarn*.

Po splnění předchozích podmínek je potřeba v adresáři `seasm` spustit příkaz `yarn install`, který nainstaluje potřebné balíčky.

Překladač se spouští příkazem `yarn run seasm cesta/k/souboru.asm`. k dispozici jsou pouze přepínače pro změnu úrovně logování procesu překladu, a to `-d` pro úroveň `debug` a `-t` pro úroveň `trace` tyto přepínače jsou určeny především pro vývoj překladače.

Výsledný binární soubor se stejným jménem, jako je vstupní soubor, zato příponou `.bin`, bude uložen do adresáře, ve kterém byl překladač spuštěn.

D.2 Popis souborů *SEASM*

Překladač zpracovává strukturované textové soubory v tomto formátu:

```
1  # Komentar
2  FUNCTIONS:
3    function jmeno_funkce
4      INSTR_S_ARGUMENTEM 0x0000 # komentar na radku s instrukci
5      INSTRUKCE
6    end function
7  PROGRAM:
8    :nazev_navesti
9    INSTR_S_ARGUMENTEM 0x0000
10   INSTRUKCE
```

Soubor je rozdělen na dvě hlavní sekce – `FUNCTIONS`, ve které je možné vytvářet funkce a `PROGRAM`, do kterého jsou zapsány instrukce hlavního programu.

Všechna klíčová slova i identifikátory jsou zpracovány bez ohledu na malá a velká písmena (tzv. *case insensitive*). Stejně tak jsou ignorovány přebytečné netisknutelné znaky kolem všech klíčových slov a identifikátorů.

Pokud se během zpracování souboru narazí na chybu, je proces překladu ukončen s chybou a do příkazové řádky je vypsáno oznámení o chybě, včetně čísla řádku, na kterém byla chyba nalezena.

D.3 Přehled klíčových slov *SEASM*

V tabulce jsou vypsána všechna klíčová slova, které *SEASM* podporuje. U instrukcí, které vyžadují argument, je označeno, jaký konkrétně požadují. *SEASM* aktuálně nepodporuje pro instrukce skoku možnost zadat číslo adresy, je nutné vždy použít název existujícího návěští.

Návěští v aktuální verzi nejsou globální a slouží ke skokům jen v rámci hlavního programu nebo dané funkce.

Klíčové slovo	Popis
`:název_návěští`	Dvojtečka označuje vytvoření návěští, neprovádí žádnou operaci; nesmí existovat více ukazatelů na stejnou pozici
`ADD`	Instrukce – TOPD1 = TOPD1 + TOPD0, v TOPD0 budou uloženy příznaky
`AND`	Instrukce – TOPD0 = TOPD0 AND TOPD1, v TOPD0 budou uloženy příznaky
`BP`	Instrukce – Pokud je aktivní ladicí režim, zastaví vykonávání programu, jinak se chová jako NOP
`CALL název_funkce`	Instrukce – Skok na zadanou instrukci
`DIS`	Instrukce – Zahodí slovo v TOPD0
`DUP`	Instrukce – Duplikuje TOPD0
`end function`	Povinné ukončení každé funkce, neprovádí žádnou operaci
`function název_funkce`	Deklarace funkce, neprovádí žádnou operaci
`FUNCTIONS`	Blok funkcí, pouze jeden v celém souboru
`IM16 16_bitové_číslo`	Instrukce – Vloží 16bitové číslo do TOPD0
`IM8 8_bitové_číslo`	Instrukce – Vloží 8bitové číslo do TOPD0
`LOAD`	Instrukce – Čtení dat z datové paměti na adrese z TOPD0 do TOPD0
`LSHIFT`	Instrukce – TOPD0 = TOPD0 << 1, v TOPD0 budou uloženy příznaky
`NOP`	Instrukce – Žádná operace, „čekání“
`NOT`	Instrukce – TOPD0 = NOT(TOPD0), v TOPD0 budou uloženy příznaky
`NZB název_návěští`	Instrukce – Podmíněný skok na zadanou adresu, provede se pokud TOPD0 != 0
`OR`	Instrukce – TOPD0 = TOPD0 OR TOPD1, v TOPD0 budou uloženy příznaky
`PERIFRD`	Instrukce – Čtení dat z registru periferie na adrese v TOPD0 do TOPD0
`PERIFWR`	Instrukce – Zápis dat z TOPD0 do registru periferie na adrese v TOPD1
`PROGRAM`	Blok instrukcí hlavního programu, pouze jeden v celém souboru
`RET`	Instrukce – Vyskočení z rutiny
`RSHIFT`	Instrukce – TOPD0 = TOPD0 >> 1, v TOPD0 budou uloženy příznaky
`STORE`	Instrukce – Zápis dat z TOPD0 na adresu z TOPD1 v datové paměti
`SUB`	Instrukce – TOPD1 = TOPD1 – TOPD0, v TOPD0 budou uloženy příznaky
`UB název_návěští`	Instrukce – Nepodmíněný skok na zadanou adresu
`XOR`	Instrukce – TOPD0 = TOPD0 XOR TOPD1, v TOPD0 budou uloženy příznaky
`ZB název_návěští`	Instrukce – Podmíněný skok na zadanou adresu, provede se pokud TOPD0 == 0